COMPUTE!'s Beginner's Guide to Assembly Language on the TI-99/4A

Peter M. L. Lottrup

COMPUTE!™ Publications, Inc. abc
One of the ABC Publishing Companies
Greensboro, North Carolina

Printed in the United States of America

ISBN 0-942386-74-4

10 9 8 7 6 5 4 3 2 1

COMPUTE! Publications, Inc., Post Office Box 5406, Greensboro, NC 27403 (919) 275-9809, is one of the ABC Publishing Companies and is not associated with any manufacturer of personal computers. TI-99/4A is a trademark of Texas Instruments, Inc.

Contents

Foreword

Machine language of computers is nothing more than a series of numbers. Those numbers are what make the computer do what you want it to do. Because it's the language of preference of your TI-99/4A, programs written in it run far faster than those written in BASIC. And because you can talk to the computer directly, without going through a translator like BASIC, you can create more powerful programs than you can with BASIC.

You write machine language (ML) programs with an *assembler.* The instructions you give to it are *assembled* (hence the term assembly language) and thus produce an ML program. But writing programs in assembly language can be difficult for the beginner. Instructions, operands, and directives can be confusing, even intimidating.

That's why you'll find *COMPUTE!'s Beginner's Guide to TI Assembly Language* such a valuable book. It's not a complete reference guide—though it includes dozens of insights, hints, and techniques on assembly language programming—but then it's not meant to be. Instead, this book takes you step by step through the process of creating and writing your own assembly language programs. Starting with a clear and easy-to-follow explanation of just how the *Line-by-Line Assembler* operates, working through such things as programming sound, sprites, and redefined characters, and ending with a complete high-resolution drawing program, you'll learn as you program. It's a hands-on approach, one that will surely take you from novice to intermediate assembly language programmer quickly and painlessly.

When you've finished the book, you'll have dozens of assembly language routines in your software library. With minor modification (and most of those modifications are outlined), you'll be able to use those same routines in your own programs. There are even several complete programs here for you to type in and run. From artist's high-resolution sketch pads to automatically moving sprites, these programs show you how powerful and fast assembly language really is.

You'll also learn how to link assembly language programs with BASIC programs, how to display register values on the screen, and how to save memory. The techniques and tricks of assembly language programming are fully covered.

Other than *COMPUTE!'s Beginner's Guide to TI Assembly Language,* all you need to begin programming in assembly language is the *Mini Memory* cartridge, and the *Line-by-Line Assembler* that comes with it. This package is still available in stores. If you want to learn assembly language programming on your TI-99/4A, it's an excellent investment.

With clear explanations and example after example for you to try out, *COMPUTE!'s Beginner's Guide to TI Assembly Language* helps you access the power of your computer's native tongue.

Introduction
Starting Off

A General Overview

When the *Editor/Assembler* package appeared on the market for the Texas Instruments 99/4A Home Computer, eager users were able to write their own assembly language programs. Assembly language is the preferred computer language of many programmers, for it allows extraordinary speed and efficiency. That's because assembly language programs create machine language code, which works directly with both the TMS9900 microprocessor, the heart and brain of the TI, and the TMS9918 Video Display Processor. Unfortunately, not all TI users could take advantage of this package. People owning the basic configuration of the computer could not use the *Editor/Assembler,* which needed the 32K memory expansion and a disk drive.

Soon after the *Editor/Assembler* was released, the *Mini Memory* cartridge became available, with all its fantastic possibilities. It allows you to read and store values in CPU and VDP memory from BASIC programs, link to assembly language programs or subroutines (optionally passing string, numeric, and array variables between the linked programs), and came with the *Line-by-Line Assembler.* The *Assembler* gave you the tool you need to create your own assembly language programs. Also included were a *debugger,* to help troubleshoot your programs, and a built-in battery. This battery made it possible for the cartridge to retain BASIC programs, BASIC files, or assembly language programs, even with the computer's power switched off and the module removed from the console.

With this *Mini Memory* cartridge, and a cassette recorder and connecting cable, the beginning assembly language programer was ready to write programs.

But both these packages lacked information you needed to learn *how* to program. The *Editor/Assembler* came with a thick manual, far too technical for the beginner to understand, and the *Mini Memory* cartridge came with two thin manuals, in which the most frequent comment told you to look up the information in the *Editor/Assembler* manual.

The beginning assembly language programmer needed a step-by-step guide to assembly language programming on the TI-99/4A. But there was no such book.

COMPUTE!'s Beginner's Guide to TI Assembly Language was written exactly for that reason, to help the beginner with assembly language programming. All examples and programs in the book have been carefully chosen and written for the *Line-by-Line Assembler,* but the basic ideas can be applied to the *Editor/Assembler* and even other assemblers. On most occasions, unless it's absolutely necessary, long and technical explanations are avoided.

Even more importantly, numerous example programs are provided, fully explained and documented. The best way to learn assembly language programming is to sit down at your computer and try everything yourself. That's what the example programs let you do.

If you have no idea whatsoever of machine language and assembly language (both are essentially the same thing—the terms are often used synonymously), make sure you read this introduction and the next four chapters carefully before going on. Then, according to what interests you most, you can turn to the appropriate chapter. These later chapters have been divided into different sections, each concentrating on one topic, such as creating sprites, generating sounds, or defining characters. They'll show you how to create programs in those areas.

I'm sure this book will be as useful to you as it was to me as I wrote it. You'll soon be creating exciting assembly language programs yourself.

All About Assembly Language

Writing a program in assembly language is like writing a program in the computer's mother tongue. When an assembly language program is run, the computer doesn't waste time translating each instruction into its own code first and then executing it (like it does when it runs BASIC programs). It means, of course, that you have to learn a new language, but the results are certainly worth it. You can see this just by comparing any BASIC program with an assembly language program. The speed and power of assembly language programs are impressive, to say the least.

Your computer works with numbers. Assembly language programs on the TI consist of a list of hexadecimal numbers

(numbers in base 16), and each number, or group of numbers, means something. (Although the listings you can see when working on a program are in hexadecimal, the computer is really working in binary, or base two.) For example, the following numbers perform the assembly language equivalent of a BASIC *CALL CLEAR*:

```
04C0
0201
2000
0420
6024
0580
0280
0300
16FA
```

Don't let this list of strange numbers scare you. You won't need to write your programs like this, thanks to something called an *assembler.* An assembler is a program (written in BASIC or assembly language) which understands a list of instructions and translates each of these instructions into its equivalent hexadecimal number. In other words, instead of *04C0* you would enter *CLR R0.*

Once you learn to use this list of instructions, writing a program in assembly language is similar to writing it in BASIC. You get an idea for a program, then sit down and write it.

There are many kinds of assemblers. The *Editor/ Assembler,* for instance, waits until you're finished writing your symbolic program and then translates it to numbers (assembles it) at your command. This allows you to keep a copy of the original program you wrote for corrections. After all, it's much easier to understand *AI R3,2* than *0223* and *0002.*

The *Line-by-Line Assembler* included with the *Mini Memory* cartridge works in a different way. When you press ENTER after typing a statement, the translation to hexadecimal is performed immediately. You can actually see this taking place. In other words, the statements are assembled one at a time, line by line, unlike the *Editor/Assembler,* which assembles the whole program at one time. Syntax errors are reported immediately and rejected by the *Line-by-Line Assembler.*

This immediate assembly is useful; you can actually see what the computer is doing with your work. However, it includes a major disadvantage. You can't save the original listing (often called the *source code*) for corrections and documentation. All you have left is the assembled program. Inserting program lines is practically impossible, and corrections are difficult to make. Thus, with this *Line-by-Line Assembler,* the best policy is to write programs on paper first, then try them on the computer. Another good idea is to divide your program into sections or blocks, each of which does one thing. For instance, a program could be divided into blocks which:

1. Clear the screen
2. Color screen black
3. Define characters

And so on.

Then, write each routine and test it until you're sure it works correctly. Finally, put them together to form a complete program. If you don't do this, you might face a 20-page program that has an error and not have the least idea where that error might be. (We'll look into debugging and programming hints in a later chapter.)

Command and Control

Writing programs in assembly language gives you complete control over most of the components of the computer, including the CPU (Central Processing Unit) RAM (Random Access Memory) and the VDP (Video Display Processor) RAM. Routines stored in ROM (Read Only Memory) and GROM (Graphics Read Only Memory) can also be accessed and used. You can modify and use values in CPU RAM directly, but to access VDP RAM you use utilities (similar to BASIC subroutines) that allow you to read from and write to VDP memory, which includes the screen information, the color tables, sprite tables, character tables, and so on.

Assembly language doesn't limit you; there is always some way to achieve the desired results. Many times there are several ways to get the same result.

With *COMPUTE!'s Beginner's Guide to TI Assembly Language,* you'll quickly be up and running with fast and powerful assembly language programs. All you have to do is turn the page.

Chapter 1

The First Step

The First Step

Loading the *Assembler*

The *Mini Memory* module allows you to load and run assembly language programs. You can even have several programs, as many as the module's memory will allow, loaded simultaneously. In fact, three programs are loaded when you load the *Line-by-Line Assembler*: the OLD option of the *Assembler*, the NEW option of the *Assembler*, and the demonstration program LINES. These three programs fit in the module's approximate 4K of memory.

To load these programs, more specifically the *Assembler*, first insert the *Mini Memory* cartridge in the computer and select option *(2) EASY BUG* from the main menu. When the title screen appears, press any key and type L. This indicates you want to load a program from tape into the *Mini Memory* module. Load the programs (NEW, OLD, and LINES) as you would normally do in BASIC. When the loading process is complete, press the FCTN and equals (=) keys at the same time. This executes the TI's QUIT command. The programs won't be erased, thanks to the module's special RAM memory. Return to the selection list and choose *(3) MINI MEMORY.* When the *Mini Memory* option list appears, choose *(2) RUN.* You'll be asked for the program name. Typing LINES and pressing ENTER execute the graphics demonstration program. For the *Line-by-Line Assembler,* choose NEW (or OLD if you are continuing a previously started program). Pressing ENTER starts the execution of the *Assembler* program. The computer enters the 40-column text display and you're ready to begin.

Understanding the *Assembler*

When you run the *Line-by-Line Assembler,* you'll see the program name and copyright on the center of the screen, and below that the following:

```
7D00  045B ■
```

The two hexadecimal numbers preceding the cursor tell you two things: the position in memory and the contents of that position. The first number, 7D00, is the memory location or address you're presently at. This is the default starting address for the *Assembler*. Most of the memory before location 7D00 is used by the *Assembler* (from approximately 71A6 onwards). That's why, though the *Mini Memory* has around 4K bytes of memory, you can only use approximately 770 bytes for your programs. The remaining memory is used for the *Assembler* itself.

The value 045B is what's currently stored in memory location 7D00, and represents an assembly language instruction. Actually, 045B is an instruction of the program LINES, loaded together with the *Assembler*. The program LINES starts at location 7CD6 and runs to location 7FB2. When writing your own programs, you'll write over the LINES program. If you type in a new instruction, the 045B will be replaced by the hexadecimal translation of the new instruction, and you'll be immediately ready to enter the new line. You can imagine the memory locations to be like BASIC line numbers.

Let's try something. Press the space bar once and then type:

```
CLR R5
```

Leave a space between *CLR* and *R5*. Note that the value 045B changes as soon as you press ENTER. The screen should now look like this:

```
7D00 04C5      CLR R5
7D02 C101 ■
```

The value 04C5 in location 7D00 is the hexadecimal translation of the instruction CLR R5. The counter has advanced to 7D02, waiting for your next instruction. The value C101 is also a machine language instruction from the program LINES which will be overwritten as you go along.

Why did you have to press the space bar before typing CLR R5? Each assembly language statement must be entered a certain way. Each line is divided into four sections, called *fields*. In each field, the computer expects to find specific information. Some fields are optional—it may not be necessary to write any information into it—others *must* be used. To exit one field and enter the next, press the space bar once. If you're not going to write anything in a field, hitting the space bar moves the cursor to the next field.

The four fields are:

- The label field
- The instruction (opcode) field
- The operand field
- The comment field

Here's a short explanation of what each field is used for, what it can contain, and whether it's optional or required.

The label field. *Labels* are used in assembly language to identify a certain memory location. If you wanted, for example, to jump to memory location 7F30, you could put a label in 7F30 and then jump to that label. Labels can be one or two characters long when you're using the *Line-by-Line Assembler.* They should be used as little as possible, since each label eats up four bytes of memory. Some label examples might be *G, LB, H1,* and so on. If one-character labels are used, the character *must* be alphabetic. Two-character labels must have an alphabetical first character and an alphanumeric second character. Labels should be used only in certain memory locations, those that need some sort of identification. Thus, the label field is certainly optional.

If you want to include a label, type it and press the space bar.

```
7D02 C101      AQ  ■                 (AQ is the label)
```

The cursor enters the next field. If no label is to be used, just hit the space bar and the cursor moves three spaces, to the second field, the instruction (opcode) field.

The instruction (opcode) field. The instruction (often called the opcode) field is where the actual instruction (or directive—see Chapter 2) is typed in. This instruction is called the *opcode* and has one to four characters. The opcode you enter is a *mnemonic* which represents the operation you want the computer to perform. For instance, *A* represents *Add.* Obviously this field must be included, since some instruction has to be given. Type in the instruction and press the space bar to enter the third field, the operand field. For example, you could enter:

```
7D02 C101      AQ  LI  ■             (LI is the instruction, or
                                      opcode)
```

The operand field. In this third field, you enter the values the opcode has to work with, which are called *operands.* This field is optional, since some instructions do not require operands. When more than one operand is used, they're separated by a comma.

Pressing the space bar moves the cursor to the comment field. If no operands are used, either press the space bar to enter the comment field or press ENTER to assemble that line.

You could type:

```
7D02 C101     AQ  LI R5,2          (R5 and 2 are the operands)
```

The comment field. The comment field works in a way similar to the tailing REMark in Extended BASIC. It's ignored by the *Assembler* and is really only your guide as you write your programs. The comments are not included when the program is translated to assembly language. This field is optional. Pressing ENTER ends the line. An example might be:

```
7D02 C101     AQ  LI R5,2          THIS IS AN EXAMPLE
```

In the above, AQ is the label, LI the opcode, R5,2 the operands, and THIS IS AN EXAMPLE the comment. Another example of a completed line could be something like:

```
7D06 E101         SWPB R1
```

At location 7D06, which had value E101 originally, no label was used, the opcode is SWPB, and only one operand (R1) is included. No comment was added.

Correcting Errors

If an error occurs while you're entering a line, one of the following two messages appears:

ERROR
or
R-ERROR

The *ERROR* message indicates a syntax error, such as writing a nonexisting instruction in the opcode field or forgetting required spaces. When this message appears, pressing FCTN-3 (ERASE) will erase the entire line so that you can start over.

If you type in a label and make a mistake, RH instead of RN, for instance, you can either erase the whole line (with FCTN-3) or type the correct label immediately after the wrong one. The *Assembler* considers the correct label as the *last two characters* entered in the field. For example, in

```
7D00 045B     AWNPG1 ■
```

the *Assembler* considers the correct label to be G1.

The same method can be used when typing in a hexadecimal (base 16) number. If you make a mistake, just type the right number after the wrong one. This time, the *Assembler* considers the last four digits as the correct number. In

```
7D00 045B     DF  AORG    >7EF87FF0
```

the *Assembler* considers >7FF0 as the correct number. The greater than symbol (>) indicates a hexadecimal number. Numbers without the > symbol are considered decimal numbers. Since the memory location and its contents are *always* in hexadecimal, the > symbol is not included before either of those numbers.

For all other error conditions, it's best to clear the whole line and type it over again.

The *R-ERROR* appears when you're trying to jump to a place in memory too far away. This message indicates an out of range error. We'll examine this error, its causes and corrections, a bit later.

Words and Bytes

Before we continue, you should understand the difference between a memory *word* and a memory *byte*.

A memory word is a four-digit hexadecimal number. An example would be the hexadecimal translation of an instruction, such as *045B*.

A word is formed by two bytes, the left or *most significant* byte and the right or *least significant* byte. If the memory word is 045B, the most significant byte is 04 and the least significant byte is 5B (remember, we're always talking about hexadecimal numbers). Many assembly language instructions use the left byte, others the right byte.

A convenient instruction lets you switch bytes in a word. This instruction is *SWPB* (SWaP Bytes). If a word is 5C97 and you use the SWPB instruction to change its bytes, the word would become 975C. You'll see how useful this instruction can be in many of the example programs.

Many instructions in assembly language work with words, and they are called *word instructions;* others work with bytes, and are called, oddly enough, *byte instructions.* You'll choose the appropriate instruction based on what you need.

Finally, note that the maximum value that can be represented by a byte is 255 (decimal), which is >FF in hexadecimal, and that the maximum value that can be represented by a word is >FFFF (65535 decimal).

Using Registers

Instead of using variables to store values, as in BASIC, in assembly language you use 16 *workspace registers.* In each of these registers you can store a one-word value. Register contents can be manipulated, just like variables. For example, the following line loads register number 7 (R7) with the decimal value of 300:

```
7D00 045B          LI R7,300
```

The LI instruction means Load Immediate and tells the computer to load the value of 300 into register 7. When you press ENTER, you'll see the following:

```
7D00 0207          LI R7,300
7D02 012C
7D04 0A54  ■
```

LI was translated by the *Assembler* as 0207; and the value of 300, to be loaded into R7, was translated as >012C (which is 300 decimal written in hexadecimal).

In other words, you can see that the instruction LI R7,300 was translated by the *Assembler* to 0207 and 012C, two words or four bytes. You'll find that, in general, most assembly language instructions occupy two or four bytes when translated into machine language and that the 770 bytes you have to work with are really more than they might seem.

Values in registers can be added, subtracted, multiplied, and divided, just like variables can be manipulated in BASIC. You'll see how this works when we begin to examine some program examples.

Just one more word must be said before going on. The values in each of the 16 registers must be stored somewhere in memory. The usual place to store them is from memory location >70B8 to memory location >70D7, though you can choose some other area.

There's an instruction which loads the memory area where the registers store their values, called LWPI (Load Workspace Pointer Immediate). All you have to do is enter LWPI in the opcode field and the memory location where the registers will begin storing their values. This location should be somewhere in the beginning of your program. The LWPI instruction isn't always needed, but it's safest to include it.

For example, to tell the computer to store the values in the workspace registers from >70B8 onwards, you'd enter:

```
7D00 02E0          LWPI >70B8
7D02 70B8
7D04 0A54  ■
```

Though you can use other memory areas to store the register values, be careful your program doesn't overwrite these locations, or their values will be forgotten by the computer. It's best to use the usual area of >70B8 onwards and avoid putting any part of your program in those addresses.

Instructions and Directives

In the opcode field of a line, you can either write an *instruction* or a *directive.* Assembly language instructions perform only one operation, like LI (Load Immediate), which loads a value into a register, or SWPB (SWaP Bytes), which exchanges the bytes in a word. The instructions for both the *Editor/Assembler* and the *Line-by-Line Assembler* are practically the same, and a list of them can be found in Appendix B.

Assembly language directives are similar to BASIC subroutines in that they perform an entire set of predetermined instructions. When the *Assembler* encounters a directive, it performs this set of preprogrammed instructions. The *Editor/Assembler* has 28 directives to work with, while the *Line-by-Line Assembler* has only 7.

Let's take a look at those seven directives of the assembler found in the *Mini Memory* module. They're very important to assembly language programming, and it's vital you know how each works.

Chapter 2

Directives and Your First Programs

Directives and Your First Programs

As mentioned in Chapter 1, the *Line-by-Line Assembler* has seven directives which you can use. They are: AORG, END, SYM, EQU, DATA, BSS, and TEXT. Each is explained in detail below.

The AORG Directive

The AORG (Absolute ORiGin) directive helps you move from one memory location or address to another, a process which is useful in correcting errors, adding information, and reviewing data. For example, when you select the NEW option of the *Mini Memory* module, you're placed at the default memory address of >7D00. If you want to start your program at some other memory location, such as >7E00, you could enter:

```
7D00 045B          AORG >7E00
```

and press ENTER. You'll then be at the correct place in memory to continue with:

```
7E00 04C3 ■
```

Remember that the directive must be typed in the opcode field, so press the space bar twice before typing AORG and again before typing the hexadecimal memory location (>7E00) to leave the instruction field and enter the operand field.

The AORG directive lets you move freely around memory, but if the memory address specified with the directive is an *odd* number, the location will be rounded down to the previous *even* number. This is because memory locations always increase by twos. If you entered the following, for instance, you could see this. Specifying >7D03 sends you to >7D02.

```
7D00 045B          AORG >7D03
7D02 C101 ■
```

In the examples which follow, you'll be entering directives beginning at location >7D00. Use AORG to return to this location when necessary.

END Directive

The END (end program) directive is used when you wish to exit the *Assembler*. The directive has no effect on the program itself and doesn't stop program execution, as is the case in

BASIC. All it does is to return to the *Mini Memory* title screen. Just type END in the instruction field, as illustrated below:

```
7D00 045B          END
```

When you press ENTER, the following message is displayed:

```
0000 UNRESOLVED REFERENCES
```

This message indicates that all labels in the operand field have actually appeared in the label field. That is, the program's not trying to jump to a nonexisting label or trying to use a value stored at a label which does not exist. If you're told that there are no unresolved references, press ENTER twice more and you'll return to the *Mini Memory* menu.

If there *are* one or more unresolved references and you exit the *Assembler*, the program will not work correctly. When you see a message indicating unresolved references, press any key *except* ENTER and you'll return to the *Assembler*. You'll be returned to the location from where you typed END. Find the unresolved references using the SYM directive (explained shortly), correct the error, and END the program once again. Continue the process if necessary until there are no unresolved references.

The SYM Directive

The SYM (Symbol Table Display) directive shows you a list of resolved and unresolved labels used in your program. To use this directive, type SYM in the instruction field and press ENTER, as in:

```
7D00 045B          SYM
```

If no labels have been used, nothing will happen. Now enter the following line at >7D00:

```
SN JMP RQ
```

Then type:

```
7D02 C101          SYM
```

You should see:

```
RESOLVED REFERENCES
SN - 7D00
UNRESOLVED REFEERENCES (JUMP)
RQ - 7D00
```

This shows that there is one *resolved reference* (a label in the label field), *SN*, in location >7D00. Any jump or reference to that label will be valid, or considered a resolved reference. However, there's also an unresolved JUMP reference, because in >7D00 the program is trying to JuMP to a nonexistent label, *RQ*. JMP is equivalent to the BASIC GOTO statement; it's an unconditional jump. Now add the following line to the above program:

```
7D02 04C0      RQ  CLR R0
```

You'll see:

```
7D00*1000
```

Type another SYM in the next line:

```
7D04 0A54          SYM
```

And you'll see this on the screen:

```
RESOLVED REFERENCES
RQ - 7D02     SN - 7D00
```

Notice that since you added a line labeled *RQ* (address >7D02), the reference is considered resolved. This is even clearer when the Symbol Table is displayed.

In a program, you're allowed to reference an as yet undefined label. When you do this, an *R* appears between the memory location and its contents. Later on, when you define this label (add the label in the label field), an asterisk is printed for each resolved reference, along with the corresponding memory location.

Let's try this out. Add the following line to the program:

```
7D04 0207          LI R7,TX
```

Pressing ENTER displays this on the screen:

```
7D06R0000
```

Notice the *R* character between the location and its contents.

Once again display the Symbol Table:

```
7D08 XXXX          SYM          (XXXX is any number at that
                                 location)
RESOLVED REFERENCES
RQ - 7D02 SN - 7D00
UNRESOLVED REFERENCES (WORD)
TX - 7D06
```

Again, in >7D04 there's a reference to a nonexistent label, *TX*, though not in a JuMP instruction. That's why, this time, the unresolved label is placed in the word references instead of the jump references.

When writing assembly language programs, it's very common to reference a label and then forget to add that label to the program. Ending a program like this would cause a program bug, or mistake. The UNRESOLVED REFERENCES message displayed upon exit from the *Assembler* reminds you if there are still undefined labels in the program. And the SYM directive helps you find those unresolved labels and correct them.

One final note: If a label is referenced in more than one location, a maximum of 32 references to that label are displayed.

The EQU Directive

The EQU (Equate), or assembly-time constant definition, directive is similar in function to the equals sign (=) in BASIC. If you want a label to be equal to a certain value, use the EQU directive. For instance,

```
7D00 045B      AB   EQU >7FE8
```

makes label *AB* equal to >7FE8.

A value assigned to a label can be assigned to another label, as in:

```
7D00 045B      NH   EQU AB
```

The above line makes label *NH* have the same value as label *AB*.

Once you've assigned a value to a label, there's no directive available to change it. Enter these lines:

```
7D00 045B      T7   EQU 118
7D00 045B      T7   *ERROR*
```

When you press the space bar after typing *T7* the second time, the error message appears. T7 already has a value, and since it can't be changed, a new value cannot be assigned to it.

The DATA Directive

The DATA, or word initialization, directive places values in the memory locations you're currently at. These values might be different data tables, character definitions, and so on. Some examples might be:

7D00 0000	**DATA 0**	(Place a zero in location >7D00)
7D02 3589	**DATA >3589**	(Place value >3589 in location >7D02)
7D04 0100	**DATA 259-3**	(Place value 256 [>100] in location >7D04)
7D06 XXXX	**DATA XN**	(If XN is defined, place the value of the label in >7D06; otherwise, the value is added when the label is defined.)
7D08 0001	**DATA 1,0,>34**	(Place a 1 in >7D08, a 0 in >7D0A, and a >34 in >7D0C)
7D0A 0000		
7D0C 0034		
7D0E XXXX ■		

In other words, the DATA directive is used to initialize one or more memory words to specific values. (In the example above, the memory word in >7D00 was initialized to 0; location >7D02 to >3589, and so forth.) Note that several memory locations can be initialized to specific values simultaneously, just by separating the values by commas (as shown for address 7D08 above).

The DATA directive is mainly used to place tables of values into specified memory locations. These tables usually give information about such things as sprites, colors, sounds, and custom characters. The following BASIC statement, for instance,

```
CALL CHAR (42,"FFA23491820100FF")
```

would be written in assembly language as:

```
7D00 FFA2          DATA >FFA2,>3491,>8201,>00FF
```

As soon as you enter the above line and press the ENTER key, you'd see:

```
7D02 3491
7D04 8201
7D06 00FF
7D08 XXXX ■
```

Note that the character definition has been placed directly into memory locations >7D00–>7D07. You've only placed the character definition in memory, however, not assigned it to any character like you might have done in the BASIC line.

The tables or single numbers placed in memory by the DATA directive must be in an area of memory where they will not be executed by the program. If they are, the program will probably not work correctly. Remember that if you place something like a character definition in memory, you don't want the computer to think it's assembly language instructions. The best technique is to add all DATA directives after the end of the program, where you're assured they won't be executed.

BSS Directive

The BSS (Block Starting with Symbol) directive is similar to the DATA directive. It also reserves a certain area of memory for the program to store information. BSS reserves a specified number of bytes, without setting them to any value (unlike DATA, which *does* initialize the locations). Try out the following:

```
7D00 045B          BSS 32
7D20 XXXX  ■
```

The 32 bytes from >7D00 to >7D1F have been reserved for later use. Again, you must be careful that the reserved memory will not be executed by the computer—make sure it's placed in an area of memory not used for the program's instructions.

An example of the use of the BSS directive could be when you perform the equivalent of a BASIC INPUT in assembly language. Whatever the user types in must be stored somewhere in memory. Suppose you want to accept words up to ten letters long. You could reserve a ten-byte area to store the word (each character is represented by one byte). The beginning of this block of memory is usually assigned a label, so that the program can know where to find it in memory. Here's an example:

```
7D00 045B    T3   BSS 10
7D0A XXXX    ■
```

In the above, you've just reserved a ten-byte block of memory, from >7D00 to >7D09, and labeled the block *T3*.

You might have already noticed that the memory locations where you type the instructions always increase by two (one word, two-byte increments). This is why, if you specify an odd number with the BSS directive, the number is rounded down:

```
7D00 045B        BSS 5
7D04 XXXX  ■
```

The line 7D00 above reserves only four bytes of memory.

The *Assembler* doesn't accept negative values with the BSS directive, and a 1 or a 0 returns you to the original location without reserving any memory.

The DATA and BSS directives work in a similar way—both set aside a memory area or block. The major difference is that DATA assigns values to the memory reserved and BSS does not.

The TEXT Directive

Completing the list of the seven *Line-by-Line Assembler* directives is the TEXT, or string constant initialization, directive. As its name indicates, it's used to store a character string in memory. For example, if you wanted to store the word *COMPUTER* in memory, you would enter:

```
7D00 045B        TEXT 'COMPUTER'
```

and press ENTER. Note that the text to be displayed must be enclosed in single quotes (the single quote is the only character which cannot be displayed in a text). When you press ENTER after typing in the above example, you'll see:

```
7D00 434F        TEXT 'COMPUTER'
7D02 4D50
7D04 5554
7D06 4552
7D08 XXXX  ■
```

What's happened is that the *Assembler* has converted each character to its hexadecimal ASCII code and stored that code in memory. (For a list of hexadecimal ASCII codes, see Appendix A.) In other words, the C has been represented by value >43, the O by value >4F, and so forth.

Text, then, occupies as many memory bytes as it has characters. Thus it's best to keep text to a minimum, since it uses up a lot of memory.

If the number of characters in a string is odd, the *Assembler* adds a *null byte* (byte >00) at the end of the text, so that the next memory location is even:

```
7D00 4845          TEXT 'HELLO'
7D02 4C4C
7D04 4F00
7D06 XXXX
```

After the >4F, the null byte (>00) was added.

The string placed in memory is not displayed on the screen by using TEXT; a set of instructions must be used to do so. Also, as with DATA and BSS, the code generated by a TEXT directive is not assembly language instructions. Again, make sure this code is left outside any program execution.

Accessing VDP Memory

Earlier it was mentioned that VDP memory (containing screen information, tables, and so forth) could not be accessed directly from the assembly language program. To write or read values from VDP RAM, you must use one of five system utility routines, which require certain information loaded into specific registers. These routines are VSBW (VDP Single Byte Write), VMBW (VDP Multiple Byte Write), VSBR (VDP Single Byte Read), VMBR (VDP Multiple Byte Read), and VWTR (VDP Write To Register).

The first four are detailed in this chapter.

Displaying a Single Character

To write one byte to VDP memory, you'll use the VSBW routine. Though in this example you'll write the byte on the screen (which forms part of VDP RAM), keep in mind that the byte can also be written to other areas of memory, such as data tables. The VSBW routine requires certain values loaded into specific registers (registers 0 and 1) before it can be executed.

In register 0 (R0) you must load the memory address where you want to write the byte. You'll want to write the byte on the screen, which occupies memory locations 0–767 (decimal). (The screen has 32 columns and 24 rows, 32 × 24 = 768 positions.) To work out the byte's screen location, using the BASIC row and column values as reference, do this: Count the number of lines *before* the line where you want to display the character—then multiply this number by 32. Add to this value the number of spaces on the next line to leave blank *before* the printing position and you'll have the correct memory location to load into R0.

For instance, to display a byte in screen position row 12 and column 7, you would multiply 11 (the line *before* line 12) by 32, then add the number of blank spaces *before* the column position. The total would be (11 × 32) + 6, which equals 358.

Once R0 has been loaded with the screen printing position, you must load the hexadecimal ASCII code of the character to be displayed into the left byte of register 1 using the LI (Load Immediate) instruction. Then all you do is branch to execute the VSBW routine to write the byte on the screen. To branch to the VSBW routine, use the BLWP (Branch and Load Workspace Pointer) instruction, which works in a similar way to the GOSUB statement in BASIC. In the *Line-by-Line Assembler*, you're not permitted to branch directly to a routine by its name, like this:

BLWP @VSBW (The @ means *at* and must be included before the routine name or position in memory)

as you can when using the *Editor/Assembler*, unless you equate the label to the position in memory of the routine first, with the EQU directive:

```
7D00 045B      VS  EQU >6024
7D00 045B          BLWP @VS
```

To avoid having to use the EQU directive and unnecessary labels, it's best to branch directly to the memory location where the subroutine is located. In other words,

BLWP @>6024

which means "branch to the routine stored at (@) hexadecimal memory location >6024 and execute it." When the routine has been executed, control returns to the next instruction after the BLWP. In the *Mini Memory* manual, pages 35 and 36, you'll see that the memory location of each routine has been included. You'll also find the memory locations in the module's ROM memory map.

At this point in the program, the byte will already have been printed on the screen. But before you end the program, you have to stop its execution or the computer will continue executing instructions in subsequent memory locations. You can create an endless loop condition by labeling a line and continuously jumping to that label:

NQ JMP NQ

Let's see how the program would look.

```
7D00 02E0           LWPI >70B8
7D02 70B8
7D04 0200           LI R0,367
7D06 016F
7D08 0201           LI R1,>2A00
7D0A 2A00
7D0C 0420           BLWP @>6024
7D0E 6024
7D10R10FF       NQ  JMP NQ        (This line could also be B *R11.
                                  Here, B is in the instruction
                                  field, and not a label)
7D10*10FF
7D12 XXXX           END
```

Explanation of the Program

First of all, the workspace area to be used by the registers, >70B8, was loaded with LWPI (Load Workspace Pointer Immediate) into memory location >7D00. Then, in memory location >7D04, R0 was loaded with the printing position. Note that the character will appear in the center of the screen (row 12, column 16), calculated by $((12-1) \times 32) + (16-1) =$ 367. Next, the left byte of R1 was loaded with the code of the character to be displayed (an asterisk, ASCII code >2A), and a 0 was placed in the right byte.

All was ready to execute the VDP single-byte write routine in location >6024. In >7D0C, a branch to that routine was executed. The only thing missing before ending the program was to stop execution; this was done with the endless loop in >7D10.

Executing the Program

When you've entered the program, END it and return to the *Mini Memory* title screen. Press FUNCTION = (QUIT) and return to the title screen. Select (2) EASY BUG and press any key to skip the instruction screen. When the question mark appears, type E7D00. This is telling the computer to EXECUTE the assembly language program which starts at location >7D00. Press ENTER and the asterisk will be displayed immediately. Further on, you'll see how you can give your program a name and execute it like the program LINES or from BASIC. FCTN = (QUIT) will not return control to you. Switch the computer off (the program will remain in memory), wait a few seconds, and switch it back on, selecting the NEW option of the *Assembler*. You'll be ready to continue.

If you substitute the line *B *R11* for the endless JMP loop (NQ JMP NQ), you can avoid the inconvenience of having to turn the computer off to break out of the program. This branching command, explained in greater detail in Chapter 4, will, in this case, return you to EASY BUG.

Using the VMBW Utility

The VDP Multiple Byte Write routine is similar to the single-byte write routine except that it writes multiple bytes to memory. A good example would be the values represented by text. In the description of the TEXT directive, it was mentioned that the text loaded into memory was not displayed on the screen just by using the directive. You can use the VMBW routine to display it on the screen.

The utility needs registers 0, 1, and 2 loaded with certain values in order to work. In R0, place the memory location where the bytes will start to be printed, just as in VSBW. In R1 load the location in memory where the bytes to be displayed will be found (remember, these should not be within execution of the program), and in R2 load the number of bytes to be written. Then you can branch to the VMBW utility stored starting at memory location >6028:

```
7D00 02E0          LWPI >70B8
7D02 70B8
7D04 0200          LI R0,67
7D06 0043
7D08 0201          LI R1,PQ
7D0AR0000
7D0C 0202          LI R2,18
7D0E 0012
7D10 0420          BLWP @>6028
7D12 6028
7D14 045B          B *R11
7D16 4153      PQ  TEXT 'ASSEMBLY LANGUAGE '
7D0A*7D16
7D28 XXXX          END
```

Program Explanation

The memory area for the registers is loaded into location >7D00. In R0, the initial printing position (67—row 3 and column 4) is loaded. R1 is loaded with the label where the text to be displayed will be found (label is PQ). The text itself will be added to the end of the program. The number of bytes to write, 18 to match the length of the text, is loaded into R2. In location >7D10, a branch executes the VMBW routine at >6028, and in location >7D14 the program returns to EASY BUG, just as in the previous example. Ending the program here would leave one unresolved reference, PQ, so it's added in location >7D16. Then the program ends.

Run the program as you did in the previous example, by selecting EASY BUG and typing E7D00, where the assembly language program begins.

The VSBR and VMBR Utilities

These routines have the opposite effect of the two previous ones. The VSBR (VDP Single Byte Read) routine reads one byte from a specific memory address and the VMBR (VDP Multiple Byte Read) routine reads a certain number of bytes, starting at a determined address.

The VSBR utility only requires R0 to be loaded with the memory address from where to read the byte. When you branch to the routine (found in location >602C), the value of the byte in that location is placed in the left byte of R1. For instance:

```
7D00 02E0          LWPI >70B8
7D02 70B8
7D04 0200          LI R0,300
7D06 012C
7D08 0420          BLWP @>602C
7D0A 602C
      .
      .
      .
```

This program segment places the value of the byte found in VDP memory location 300 into the left byte of R1.

The VMBR routine requires R0 to be loaded with the VDP RAM memory address from where to start reading the bytes, R1 loaded with the place in memory where to put these bytes (an area reserved with the BSS directive), and R2 with the number of bytes to be read. A branch to the VMBR routine in location >6030 does the rest. Here's a sample program segment to show you how it can be done.

```
7D00 02E0            LWPI >70B8
7D02 70B8
7D04 0200            LI R0,>0585
7D06 0585
7D08 0201            LI R1,BF
7D0AR0000
7D0C 0202            LI R2,10
7D0E 000A
7D10 0420            BLWP @>6030
7D12 6030
7D14 045B            B *R11
7D16 XXXX      BF    BSS 10
7D0A*7D16
7D20 XXXX            END
```

This loads the memory area labeled BF with the ten bytes read from VDP RAM locations >0585 on up. The memory area BF has been added where it will not be executed as program instructions (stored in memory address 7D16). As with the two previous sample programs, this one returns you to EASY BUG.

Since this segment does not write to any screen memory addresses, nothing appears to happen when the program executes. The next example, however, will visually demonstrate the VMBR routine by writing what was read *from* the screen back *to* the screen.

Using VMBW and VMBR

The last example in this chapter combines the VMBW (VDP Multiple Byte Write) and VMBR (VDP Multiple Byte Read) routines to place a message on the screen, read it, and print a portion of it elsewhere. Type in the following program:

Read and Write

```
7D00 02E0            LWPI >70B8    (Load memory area for
                                    registers)
7D02 70B8
7D04 0200            LI R0,67      (Memory location where bytes
                                    will be written)
7D06 0043
7D08 0201            LI R1,PQ      (Bytes to be written are stored
                                    at PQ)
```

```
7D0AR0000
7D0C 0202        LI R2,18        (Number of bytes to write)
7D0E 0012
7D10 0420        BLWP @>6028 (Execute the VMBW routine)
7D12 6028
7D14 0200        LI R0,67        (Memory location where bytes
                                 will be read)
7D16 0043
7D18 0201        LI R1,BF        (Read in bytes will be stored at
                                 BF)
7D1AR0000
7D1C 0202        LI R2,18        (Number of bytes to be read in)
7D1E 0012
7D20 0420        BLWP @>6030 (Execute the VMBR routine)
7D22 6030
7D24 0200        LI R0,330       (Memory location where bytes
                                 will be written)
7D26 014A
7D28 0201        LI R1,BF        (Bytes to be written are stored
                                 at BF)
7D2AR7D1A
7D2C 0202        LI R2,8         (Number of bytes to be written)
7D2E 0008
7D30 0420        BLWP @>6028 (Execute the VMBW routine)
7D32 6028
7D34 045B        B *R11          (Return to EASY BUG)
7D36 5345    PQ  TEXT 'ASSEMBLY LANGUAGE '
7D0A*7D36
7D48 10E2    BF  BSS 18          (Set aside 18 bytes for storing
                                 message)
7D2A*7D48
7D1A*7D48
7D5A XXXX        END
```

When executed with EASY BUG (E7D00), the program will write the message ASSEMBLY LANGUAGE on the top of the screen, read the entire message from the screen, and then print the first eight characters (bytes) at screen location 330.

Saving Your Program on Tape

Whenever you want to save your program on tape, select the EASY BUG option from the master selection list, press any key to skip the instructions, and type S. This command means "save the contents from memory to tape." The computer will have to know from what memory location to start saving and up to where to continue the process. It's always best to save the entire contents of the module's 4K RAM to tape (from >7000 to >7FFF), so when the question mark appears, type S7000 and press ENTER. The compter will ask *TO ?*. Type >7FFF and press ENTER.

It's not necessary to include the greater than (>) symbol. Now follow the usual process to save a program onto tape.

To load the program from tape, follow the same instructions as for loading the *Assembler* and LINES programs. If the name of your program has been added, run the program like the LINES demonstration program or by calling it from BASIC. Otherwise, use the E (Execute) instruction of EASY BUG as you've been doing already.

You're already using the *Line-by-Line Assembler* to do complex things like read and write from your TI's screen memory. Of course, there's more to learn, more powerful assembly language programming techniques. That's what Chapter 3 is all about.

Chapter 3

More Programming Power

More Programming Power

Instructions are kept very simple in assembly language—that's both an advantage and a disadvantage, for although the instructions are easy to remember (for the most part), it does make program listings quite long. Don't be intimidated by an assembly language program's length. Just because it's long doesn't mean it's complicated. To perform even a simple operation, such as a machine language equivalent to BASIC's CALL CLEAR, a whole set of instructions has to be written.

But assembly language programs are powerful. And in this chapter, you'll see more detailed examples to help you solidify your programming knowledge.

Note: From this point on, program listings will not include the contents of the locations. You'll see the memory address, and the instruction to type in. Simply enter the instructions as you've done in the first two chapters.

Increasing and Decreasing a Value

Though instructions are provided to add and subtract values stored in memory locations and registers, four convenient instructions exist which operate directly. They are: INC, INCT, DEC, and DECT.

INC (INCrement) increases the value in a memory address or register by one; adds one to the value there:

```
7D00      INC R3           (Adds one to the value stored in
                           R3)
```

DEC (DECrement) decreases the value in a register or memory location by one; subtracts one from the value there:

```
7D02      DEC @>7F00       (Subtracts one from the value
                           stored at memory location >7F00)
```

(Remember that when you reference a hexadecimal or decimal memory location directly, it must be preceded by the @ symbol, except when using jump instructions.)

INCT (INCrement by Two) adds two to the value in a register or memory address:

```
7D04      INCT @>7E18      (Adds two to the value stored in
                           location >7E18)
```

DECT (DECrement by Two) subtracts two from the value in a register or memory location:

```
7D06      DECT R5          (Subtracts two from the value in
                           R5)
```

Use these instructions whenever you need to add or subtract one or two from a value, instead of using the addition and subtraction instructions. The latter instructions use more bytes.

Adding and Subtracting

AI. To add to or subtract from a value stored in a register, you can use the AI (Add Immediate) instruction. This instruction is called an *immediate* instruction because the first operand is a register and the second a number (decimal or hexadecimal). See Appendix B for a list of the instructions.

If you want to add, say 32 to the value in R4, you would enter:

```
7D08        AI R4,32          (The value in R4 is increased by
                              32)
```

To add >312 to the value in R12:

```
7D0C        AI R12,>312       (The value in R12 is increased by
                              >312)
```

The same instruction can be used to subtract a value from the contents in a register (nothing called *subtract immediate* exists). Just add the *negative* value of the number you want to subtract. For example, to subtract 712 from the value in R7, you would type:

```
7D00        AI R7,-712        (Subtracts 712 from the value in
                              R7)
```

To subtract >24 from the value stored in R15:

```
7D04        AI R15,->24       (Subtracts >24 from the value in
                              R15)
```

The result of addition or subtraction by the AI instruction is placed in the same register where the initial value was stored. In the previous example, for instance, the value after subtracting >24 from the value in R15 is placed back in R15.

A and **S.** In many cases you might want to add or subtract the values in two registers, two memory locations, or a register and a memory location. Then the A (Add words) and S (Subtract words) instructions are useful. These are *word instructions,* which means that they work with the complete four-digit hexadecimal number in a register or memory address.

The A (Add words) instruction adds the word value in the first operand to the word value in the second operand. It then places the addition in the second operand. Assuming that R3 is loaded with >1201 and R1 with >1362, the following line:

```
7D00      A R1,R3
```

adds >1201 and >1362 (for a total of >2563), and places the answer in R3. The first operand remains unchanged by the operation. If you want the answer in R1, just invert the operands, like so:

```
7D00      A R3,R1           (Adds the values and places the
                            addition in R1, leaving R3
                            unchanged)
```

Some more examples:

```
7D02      A R3,@>7FC0       (Adds the word value in R3 to the
                            value stored in location >7FC0
                            and places the answer in >7FC0)
7D06      A @>7FC0,R3       (Same as above, but answer is
                            placed in R3)
7D0A      A @>7D04,@>7E12   (Adds the value in >7D04 to the
                            value in >7E12, placing the an-
                            swer in >7E12)
```

The S (Subtract words) instruction works the same way, only subtracting the word values of two registers, two memory locations, or a register and a memory location. The value of the first operand is subtracted from the value of the second operand and the answer placed in the second. For example, if R5 is loaded with 2 and R7 with 5, then

```
7D00      S R5,R7
```

subtracts the value in R5 (2) from the value in R7 (5) and places the answer (3) in R7. The value of R5 remains unchanged. Other examples are:

```
7D00      S @>7FC2,@>7100   (Subtracts the word value at loca-
                            tion >7FC2 from the word value
                            at >7100, storing the answer at
                            >7100)
7D00      S @>7F00,R14      (Subtracts the word value found
                            at memory location >7F00 from
                            the word value in R14, placing
                            the answer in R14)
```

AB and SB. Two instructions similar to A and S are AB (Add Bytes) and SB (Subtract Bytes). Both do the same as the word instructions, but operate only with the left (most significant) byte of the word, leaving the right (least significant) byte unchanged. If R4 is loaded with >0492 and R5 with >1067, the instruction

```
7D00         AB R4,R5
```

adds the left byte of the word in R4 (>04) to the left byte of the word in R5 (>10), placing the answer (>14) in the left byte of R5. The right byte of R5 remains unchanged. The value found in R5 now would be >1467. R4 remains unchanged.

Here's another example:

```
7D00         AB @>7BFE,@>7100
```

This adds the left byte of the word found at (@) >7BFE to the left byte of the word at >7100. The answer is placed in the left byte of the word at >7100, and the least significant byte (>7101) remains unchanged.

SB (Subtract Bytes) works in the same way as S, but subtracts the value found in the left byte of the word in the first operand from the left byte of the word in the second operand. The resulting answer is stored in the left byte of the word in the second operand. If R2 is loaded with >0127 and R1 with >0256, the instruction

```
7D00         SB R2,R1
```

subtracts >01, the left byte of R2, from >02, the left byte of R1. R1 now contains >0156, while R2 remains unchanged.

Instruction Formats

Each instruction is classified into one of nine formats. For example, all instructions which use two operands in the operand field, separated by a comma, and where the operands are general addresses (such as a memory address or a workspace register) are considered **Format I** instructions. They're also called "two general address instructions."

```
7D00         A @>837C,R5      (>837C and R5 are two general
                               addresses separated by a comma,
                               so A is a Format I instruction)
```

Other formats which include instructions you'll use are:

Format II. All the jump instructions, which transfer control to a memory location or a label representing a memory location.

```
7D00         JMP LP
```

Format III. Logical instructions, which contain a general address as first operand, separated by a comma from the second operand, which is a workspace register.

Format VI. Single address instructions, which require only a general address. Examples include the INC, INCT, DEC, and DECT instructions discussed earlier in this chapter.

Format VIII. Immediate instructions, which require a register as the first operand, followed by a comma and a numeric expression in the operand field.

```
7D00          LI R5,3
```

Also included in this format are two instructions requiring only a numeric expression in the operand field:

```
7D00          LWPI >70B8
```

and two instructions requiring only a register in the operand field.

Format IX. Extended operation instructions. This format includes the extended operation instructions and the multiplication and division instructions.

Don't worry about understanding the formatting of instructions yet. As you start to work on your own assembly language programs, you'll get used to what instructions to use where. Whenever you come across an instruction which gives you some doubt about what kinds of operands it works with, refer to Appendix C. The instructions are listed there, as well as the operands each uses.

Comparing Values

If you want to compare the value in a register to a number, you'll use one of the compare instructions. There are three we'll look at here.

CI (Compare Immediate) is a Format VIII immediate instruction, and requires a register as the first operand and a numeric expression as the second operand. For example, to compare the value stored in R5 (register 5) to 118, you would enter:

```
7D00          CI R5,118             (Compares the value stored in R5
                                    to 118)
```

To compare the words in two memory locations, two registers, or a memory location and a register, use the C (Compare words) instruction. For instance:

```
7D00        C R3,R4           (Compares the word value in R3
                              to the value in R4)
7D02        C @>8374,R3       (Compares the word value stored
                              at >8374 to the word value in R3)
```

Finally, to compare the left bytes of two words, the CB (Compare Bytes) instruction can be used, like this:

```
7D00        CB R3,R4          (Compares the left byte of R3 to
                              the left byte of R4. If the bytes are
                              the same, the registers are consid-
                              ered equal even if the least signifi-
                              cant [right] bytes are different.)
7D02        CB @>7500, @>7C00 (Compares the left bytes of the
                              words stored in memory locations
                              >7500 and >7C00)
```

Jumping According to a Result

After having made a comparison, you'll want to transfer program control according to the result. You've already seen the JMP (JuMP) instruction, similar to BASIC's GOTO. But assembly language has other kinds of jumps which can be used according to the result of a comparison. They have the same function as the IF-THEN in BASIC. Some of these instructions are:

JEQ (Jump if EQual). If the compared values are equal, this jump will be executed. Otherwise it's ignored, and the program continues with the next instruction after the jump.

```
7D00        C R1,R2           (Compares R1 and R2)
7D02        JEQ LP            (If they are equal, transfers control
                              to address labeled LP)
```

JGT (Jump if Greater Than). If the value of the first operand is greater than the value of the second operand, execute the jump. Otherwise not.

```
7D00        CI R3,300         (Compares the word value in R3
                              to the decimal number 300)
7D04        JGT NQ            (If the value in R3 is greater than
                              300, control is transferred to
                              memory location with label NQ)
```

JHE (Jump if High or Equal). If the value of the first operand is greater than or equal to the value of the second operand, the jump executes.

```
7D00        C R3,R4         (Compares the value in R3 to the
                            value in R4)
7D02        JHE P3          (If the word in R3 is greater than
                            or equal to the word in R4, con-
                            trol is transferred to the location
                            labeled P3)
```

JLE (Jump if Low or Equal). If the value of the first operand is less than or equal to the value of the second operand, this jump executes.

```
7D00        C R7, @>7F00    (Compares the word in R7 to the
                            word stored at memory location
                            >7F00)
7D04        JLE >7D08       (If the value in R7 is less than or
                            equal to the value found at
                            >7F00, control is transferred to
                            >7D08)
```

JLT (Jump if Less Than). If the value of the first operand is less than the value of the second operand, the program executes the jump.

```
7D00        C NM,R2         (Compares the value stored in
                            location labeled NM to the value
                            in R2)
7D04        JLT A5          (If the value in NM is less than
                            the value in R2, program execu-
                            tion continues in the memory ad-
                            dress labeled A5)
```

JNE (Jump if Not Equal). If the two values compared are different, the jump is executed.

```
7D00        C R3,R4         (Compares R3 and R4)
7D02        JNE >7D50       (If the values are different, control
                            passes to location >7D50)
```

Other jump instructions exist, but the above are the most frequently used. One thing to remember is that a jump instruction cannot jump to a location more than >100 (256) bytes away. If you try to do this, you'll get an *R-ERROR* (out of range) message. To see this, try the following:

```
7D00        JMP >7F00
```

As soon as you press ENTER, an *R-ERROR* message will appear, because >7F00 is more than 256 bytes away from >7D00. To avoid this error message, it's best to use the B (Branch) instruction, which allows you to branch to *any* memory address in the program:

7D00	B @NG	(Branches to the location labeled NG. Here you must include the at [@] sign before the memory location or label)
7D04	B @>7F00	(Branches to memory location >7F00)

Branching After a Comparison

You don't have all the different kinds of jump instructions available with the branch instruction. What would happen if a conditional jump caused an *R-ERROR* because of trying to jump to a location more than 256 bytes away? For instance, consider the following error:

```
7D00        CI R2,300
7D04        JLT NG *R-ERROR*
```

If the value in R2 is lower than 300, you want the program to jump to the location labeled NG. But NG is too far away in memory to be reached by a jump (in this case, JLT) instruction. How can this same routine be done using the B (Branch) instruction?

It's not hard. Just invert the problem. Instead of comparing and looking for results less than, compare and look for results greater than. Glance at the following solution:

```
7D00        CI R2,300
7D04        JHE NQ
7D06        B @NG
7D0A   NQ
(program continues)...
```

In the first example you told the computer to jump to NQ if R2 was *less than* 300, but here you said that if R2 is *equal to or greater than* 300, skip to NQ and continue the program. If *not* equal to or greater than, it branches back to NG (>7D06).

Creating More Programs

All this theoretical background has shown you a whole new set of instructions. Now we'll write some example programs to see how many of these instructions can be used.

A Delay Loop

In most of your assembly language programs, you'll need to use delay loops to *slow down* execution. Assembly language is fast—often too fast. Many times you need to slow it down so people can use the program.

A delay loop is simple to create; one way is to load a value in a register and decrease it until it's equal to zero. It's similar to something like a FOR I = 1 TO 1000:NEXT I statement in BASIC.

```
7D00        LI R7,5000
7D04   LP   DEC R7
7D06        CI R7,0
7D0A        JNE LP
```

When you're comparing the first operand to zero (only), as in the above, you don't need to include the comparison. The previous could thus be written as:

```
7D00        LI R7,5000
7D04   LP   DEC R7
7D06        JNE LP
```

The jump (JNE) instruction automatically compares R7 (the last register operated with before the jump instruction) to zero.

Remember that the maximum value you can load in a register is 65535 (>FFFF), and in assembly language, a loop with such a delay only causes the program to pause for around a second. The following program waits with the maximum loop value and then prints the word FINISHED on the screen.

Maximum Loop—FINISHED

```
7D00        LWPI >70B8       (Load the memory area for the
                             registers)
7D04        LI R9,>FFFF      (Load delay value into R9)
7D08   LP   DEC R9           (Decrease value in R9 by one)
7D0A        JNE LP           (If not zero, return to loop LP)
7D0C        LI R0,300        (Delay loop finished. Load screen
                             printing position)
7D10        LI R1,TX         (Load position of text in memory,
                             TX)
7D14        LI R2,8          (Load the length of the text)
7D18        BLWP @>6028      (VMBW routine to display the
                             message)
```

```
7D1C        B *R11            (Return to EASY BUG)
7D1E   TX   TEXT 'FINISHED'   (Text to be displayed)
7D26        END
```

When you run this program, the delay is only a moment long. For longer delays, you'll need to use nested loops. For example, to make the previous delay five times as long, load another register with the value of five and each time a delay is executed, decrease it. When the register is zero, continue. If it's still not zero, then return to the delay loop:

Longer Delays

```
7D00        LWPI >70B8        (Load memory area for registers)
7D04        LI R12,5          (Number of times to execute outer
                              loop)
7D08   L1   LI R5,>FFFF       (Number of times to execute inner
                              loop)
7D0C   L2   DEC R5            (Decrease value of inner loop)
7D0E        JNE L2            (If not equal to zero, loop not
                              completed)
7D10        DEC R12           (Decrease value of outer loop)
7D12        JNE L1            (If not zero, return to repeat inner
                              loop)
7D14        LI R0,300         (Loops finished. Load screen dis-
                              play position)
7D18        LI R1,TX          (Load position of text in memory)
7D1C        LI R2,5           (Load length of text)
7D20        BLWP @>6028       (Branch to display text)
7D24        B *R11            (Return to EASY BUG)
7D26   TX   TEXT 'READY'      (Add text to program)
7D2C        END
```

This program creates two nested loops. The inner loop is executed five times (the value in the outer loop) before the program continues. Run the program and you'll see that the computer waits a little longer than before.

Clearing the Screen

In this next example you'll create a routine to clear the screen, located in VDP memory from locations 0 to 767.

If you didn't have the CALL CLEAR subroutine in BASIC, how would you clear the screen? The easiest way would be to print a blank character (a space) in each of the 768 screen positions. The same thing can be done in assembly language; using the VSBW (VDP Single Byte Write) routine, you can print a blank on each of the 768 positions.

Clear Screen with Assembly Language

```
7D00         LWPI >70B8       (Load memory area for registers)
7D04         CLR R0           (Load zero in register 0)
7D06         LI R1,>2000      (Load the ASCII code for the
                              space [>20] in the left byte of R1)
7D0A   LP    BLWP @>6024      (Print the blank)
7D0E         INC R0           (Increase the screen printing
                              position)
7D10         CI R0,768        (Compare it to the first position
                              beyond the screen. Screen goes to
                              location 767)
7D14         JLT LP           (Screen position is still smaller, so
                              printing is not complete. Return to
                              loop LP)
7D16         B *R11           (Return to EASY BUG)
7D18         END
```

This program uses the same instructions as previous examples, and the program explanation included beside each instruction should help you follow its workings. The only new instruction used is CLR (CLeaR) in location 7D04, which sets the word value in a register or memory address to zero, as:

```
CLR @>8374                    (Load >0000 into >8374)
```

It's better to use CLR than to LI (Load Immediate) the value of zero, because CLR uses only two bytes of memory with registers and also can be used to directly clear a memory location.

Run the screen-clearing routine once you've entered it. Keep an eye on the E7D00 message at the bottom of the screen. Note how quickly it's erased, indicating that the CALL CLEAR routine was successful.

Crossing At

The next example program makes the @ symbol run along the top of the screen, from left to right. This is another easy routine to program. You must create a loop to print the @ from positions 0 through 31, erasing it again after each print by printing a blank over it.

Moving @

```
7D00        LWPI >70B8      (Load the memory area for the
                            registers)
7D04   L1   CLR R0          (Load R0 with zero, first screen
                            position)
7D06   L2   LI R1,>4000     (Load R1 with code for the @
                            symbol)
7D0A        BLWP @>6024     (Print symbol on the screen)
7D0E        LI R1,>2000     (Load code for blank to erase the
                            @ symbol)
7D12        BLWP @>6024     (Print blank erasing the @ sign)
7D16        INC R0          (Increase printing position by one)
7D18        CI R0,31        (Is it the last position of the top
                            line?)
7D1C        JNE L2          (No. Return to print a new @)
7D1E        JMP L1          (Yes. Return to reset printing po-
                            sition and start over)
7D20        END
```

Run the program. Doesn't the @ symbol move a little bit too fast? To make the program run a bit slower, you'll have to add a couple of delay loops. Adding one after the @ symbol is printed, and another after the blank has been printed, should be enough. The new listing would look like this (if you don't want the @ to blink so much, try leaving out the delay loop after the blank is printed):

Slower @

```
7D00        LWPI >70B8      (Load memory area for the
                            registers)
7D04   L1   CLR R0          (Initial screen printing position)
7D06   L2   LI R1,>4000     (Load the code for the @ symbol)
7D0A        BLWP @>6024     (Print the @)
7D0E        LI R7,2000      (Load R7 with the value for the
                            delay loop)
7D12   L3   DEC R7          (Decrease the loop value)
7D14        JNE L3          (If R7 equals zero, the program
                            continues; if not, control returns
                            to L3)
```

```
7D16        LI R1,>2000       (Load code for the blank)
7D1A        BLWP @>6024       (Print the blank, erasing the @
                              sign)
7D1E        LI R7,2000        (Load R7 with the value for the
                              delay loop)
7D22   L4   DEC R7            (Decrease loop value)
7D24        JNE L4            (If R7 is not equal to zero, control
                              returns to L4)
7D26        INC R0            (Screen printing position is
                              increased)
7D28        CI R0,31          (Is it the last position of the top
                              line?)
7D2C        JNE L2            (No. Return to printing routine,
                              label L2)
7D2E        JMP L1            (Yes. Restart complete routine, la-
                              bel L1)
7D30        END
```

Run this new program. If you want to change the printing speed of the @ symbol, changing the values in the delay loops, you don't need to retype the program. Just return to the module's title screen, select (3) MINI MEMORY and (2) RUN. Type OLD and press ENTER. Then use the AORG directive to get to memory location >7D0E, where the value for the first delay loop was loaded:

```
XXXX AORG  >7D0E
7D0E ■
```

Type the LI instruction once again, including the value you want to use in R7:

```
7D0E        LI R7,XXX         (where XXX is the new delay
                              value)
```

Then use the AORG directive to get to the next delay loop:

```
7D12        AORG >7D1E
7D1E        LI R7,YYY         (where YYY is the new value for
                              the second delay)
7D22        END
```

End the program and run it again. One recommended change is to leave the first loop with a 2000 delay and change the second loop to contain a delay of only 2.

Squaring the Screen

This program will make the @ sign flash in a square around the screen. For the top of the screen, you'll do the same as for the previous example program, increasing the screen position with the INC instruction. Then the symbol is moved down one line at a time. For the printing position to move exactly one line down, you must add 32 characters to the current screen printing position in R0. The instruction *AI R0,32* will move the @ sign down one line at a time.

To move the symbol from right to left, you decrease the value in R0 with the DEC instruction. Then, move the symbol up again by subtracting 32 from its current position. In other words, add −32 for each line up with *AI R0,−32*. Take a look at the next program:

Squared @

```
7D00       LWPI >70B8       (Load memory area for the
                            registers)
7D04       CLR R0           (Clear register 0)
7D06  L1   LI R1,>4000      (Load R1 with the hexadecimal
                            code for the @ symbol)
7D0A       BLWP @>6024      (Print the @ symbol on the
                            screen)
7D0E       LI R1,>2000      (Load code for the blank)
7D12       BLWP @>6024      (Print blank on the screen, delet-
                            ing the @)
7D16       INC R0           (Increase printing position)
7D18       CI R0,31         (Has the last position [31] been
                            reached?)
7D1C       JNE L1           (No. Return to first printing loop)
7D1E  L2   LI R1,>4000      (Yes. Load @ code in R1)
7D22       BLWP @>6024      (Print it on the screen)
7D26       LI R1,>2000      (Load blank in R1)
7D2A       BLWP @>6024      (Print it on the screen)
7D2E       AI R0,32         (Move printing position one line
                            down)
7D32       CI R0,767        (Has last line been reached?)
7D36       JNE L2           (No. Return to second printing
                            loop)
7D38  L3   LI R1,>4000      (Yes. Load code for @ into R1)
7D3C       BLWP @>6024      (Print @ on the screen)
```

```
7D40          LI R1,>2000       (Load code for blank into R1)
7D44          BLWP @>6024       (Print blank, deleting the @ sign)
7D48          DEC R0            (Decrease the printing position)
7D4A          CI R0,736         (Has the first position of the last
                                line been reached?)
7D4E          JNE L3            (No. Stay in the third printing
                                loop)
7D50    L4    LI R1,>4000       (Yes, load code for @ in R1)
7D54          BLWP @>6024       (Print the @ on the screen)
7D58          LI R1,>2000       (Load code for blank into R1)
7D5C          BLWP @>6024       (Print blank on the screen)
7D60          AI R0,-32         (Move printing position one line
                                up)
7D64          JNE L4            (If printing position in R0 is not
                                equal to zero, stay in loop 4)
7D66          JMP L1            (Printing sequence complete. Start
                                over at L1)
7D68          END
```

This program has no delays. If you want to add them, placing one after each printed @ symbol will be enough.

The program above can be written in more efficient and/or shorter ways. There's almost always more than one way to write a routine. But with the instructions you know at this point, the best and clearest way to write this particular program is the way you just saw.

General Addressing Modes

When an instruction works with two operands in the operand field, we call the *source operand* the one we're going to operate on or manipulate. The operand where the result of the operation is placed is called, naturally enough, the *destination operand.* In the following example

```
7D00          A R4,R3
```

R4 is the source operand and R3 the destination operand.

There are five ways to work with values in a register or memory location. Called the General Addressing Modes, they are:

Workspace Register Addressing. This is what you've been doing in the example programs so far, working with the values contained in a register (from 0 to 15):

A R8,R9 (Adds the word in R8 to the word in R9, placing the result in R9)

Workspace Register Indirect Addressing. This is when the register contains the memory location where the value to be used is found. Indirect addressing is specified by preceding the register with an asterisk:

A *R3,*R4 (Adds the value found at the *address* stored in R3 to the value found at the *address* stored in R4 and places the answer in the location specified in R4)

If an asterisk precedes the register, it represents the phrase *the contents found in the memory location specified by the value in this register.* In the previous example, if R3 contained >7D00 and R4 contained >7E00, the instruction adds the word stored in >7D00 to the word stored in >7E00, placing the answer at memory location >7E00.

You don't need to precede both operands with an asterisk if you want only one of the registers to hold the address of a memory location. For instance, you could use:

S *R2,R5 (Subtracts the value stored in the location addressed by the value in R2 from the value in R5 and stores the answer in R5)

Note that the value found in register 5 was used, not the contents of a memory location loaded into that register.

You've seen the asterisk, and thus indirect addressing, used several times already. The line *B *R11,* which returned several example programs to EASY BUG, meant to branch to the memory location addressed by the value in register 11.

Workspace Register Indirect Auto-Increment Addressing. Symbolized by following the register with a plus (+) sign, this mode increases the memory address stored in the register by one byte or one word, according to the instruction used. For example, assuming R3 is loaded with memory location 7D00, the following instruction

AI *R3+,100

adds 100 to the value found in the memory address stored in R3 (the asterisk causes this), and then increases the memory address in R3 by two, leaving >7D02 stored in R3. The + does this. The increment was of one word, two bytes, because AI is an instruction operating with words.

In the next example, imagine that R5 is loaded with >7EF8, and R7 loaded with 7F50. The instruction

```
7D00          AB *R5,*R7+
```

adds the left byte of the word in memory location >7EF8, stored in R5, to the left byte of the word in memory location >7F50, stored in R7. The value in R7 is incremented by one byte because AB is a byte instruction. If this instruction was executed a second time, the left byte of the word stored in the memory location found in R5 (>7EF8) would be added to the byte found in >7F51 (stored in R7), because the value in R7 was already incremented by one byte when the instruction was executed the first time.

This addressing mode is very useful when working with data tables, as you'll see in a later chapter.

Symbolic Memory Addressing. This is when you work directly with a memory location or a label at a memory address. The symbolic memory address is preceded by the @ symbol. Some examples are:

A	**@>7F00,@>7EC2**	(Adds the word found in >7F00 to the word found in >7EC2, placing the answer in location >7EC2)
CB	**@NM,@>7100**	(Compares the left byte of the word found at the memory location labeled NM to the left byte of the word found in >7100)
SB	**R7,@>7D08**	(Subtracts the left byte of the word in R7 from the left byte of the word stored in memory location >7D08, placing the difference in the left byte of the word at >7D08)

Indexed Memory Addressing. An indexed memory address is preceded by the @ sign and followed by a register enclosed in parentheses (any register except R0 may be used). To understand this type of addressing, study the following examples:

S	**@5(R9),R7**	(Subtracts the word stored in the memory address found by adding 5 to the value in R9 from the value in R7. The difference is placed in R7)

A	**R7,@NM-3(R3)**	(Adds the word found in R7 to the word stored in the location found by subtracting 3 from the value in NM, and then adding this to the value in R3. The answer is placed in the same computed address)

Indexed memory addressing is not used as frequently as the others. We'll cover it in more detail later.

The previous modes, however, are very useful when programming. You'll see just how useful in the next chapter.

Chapter 4

The Next Few Steps

The Next Few Steps

Planning an Assembly Language Program

Assembly language programs are not difficult to write, as long as you plan them carefully. Due to the lack of editing features in the *Line-by-Line Assembler,* you'll find that writing programs at the TI's keyboard is difficult. The only exceptions are extremely short or simple routines. The best thing to do is to write your programs on paper first.

Before you start this, though, divide the program into *blocks,* as mentioned in the Introduction. For instance:

- CALL CLEAR
- Print title screen
- And so on

Then create and individually test each routine. When you're sure that all the segments work correctly alone, put them together to form the complete program.

Writing a complete program on paper and then testing it might lead to disastrous results, and may leave you staring at a 4K program, without knowing where the program bug is. Hours of time wasted.

You'll probably have to do a great deal of what I call research each time you write a new program. This research ranges from investigating memory tables to interpreting strange errors or confusing effects from the computer. Never get discouraged—it's all part of the intricate world of assembly language.

Repeated Coding: Subroutines Needed

Every byte counts when you're using the *Line-by-Line Assembler.* Memory has to be used very carefully. Routines which are used more than once in the same program should be created as subroutines, just as you often do in BASIC. The instruction to call a subroutine, similar to BASIC's GOSUB, is the BL (Branch and Link) instruction. It's used like the B (Branch) instruction, transferring control to any memory location desired. However, the instructions should not be confused.

While B sends control to another memory address (as BASIC's GOTO does), BL lets you return from a routine to the instruction immediately following the BL. That's just the way BASIC's RETURN works. The BL instruction is used like this:

```
7D00         BL @NM          (Branches and links to the sub-
                             routine starting at the memory
                             location labeled NM)
7D04         BL @>7BC0       (Branches and links to the sub-
                             routine starting at memory loca-
                             tion >7BC0)
```

When using BL, the memory location with the instruction *immediately following* that containing the BL instruction is placed in register 11. To return from a subroutine, then, all you have to do is branch to the value stored in R11:

```
7FC0         B *R11          (Branches to the address stored in
                             R11, returning to the main
                             program)
```

Thus control returns to the main program, specifically to the instruction following the call to the subroutine. This method of returning from a subroutine is very helpful because the RT (ReTurn) instruction in the *Editor/Assembler* is not available in the *Line-by-Line Assembler.* Fortunately, *B *R11* does the same thing as RT.

In the following example, the CALL CLEAR routine is implemented as a subroutine labeled CR. Each time the screen has to be cleared in a program, you'd just branch and link (BL) to this routine.

```
7D30         BL @CR          (Screen has to be cleared. Branch
                             and link to the routine at CR)
  .
  .
  .
7F80    CR   CLR R0          (Screen clearing subroutine be-
                             gins. Load R0 with zero)
7F82         LI R1,>2000     (Load R1 with the ASCII code for
                             the blank, >20)
7F86    LP   BLWP @>6024     (Print the blank on the screen)
7F8A         INC R0          (Increase printing position)
7F8C         CI R0,768       (Has last printing position been
                             passed?)
7F90         JLT LP          (No. Return to printing loop)
7F92         B *R11          (Yes. Clearing routine finished.
                             Return to instruction following the
                             subroutine call by branching to
                             the memory location stored in
                             R11)
```

Using NOP

This convenient instruction helps you prepare your programs for later editing. The NOP (No OPeration) instruction leaves one or more blank memory locations which the computer ignores as it continues to the next assembled instruction. This allows you to later correct errors in your program by adding instructions in those free memory addresses.

Consider the following example, where memory location >8374 should have been cleared for the program to work correctly. Luckily, some locations were left open just in case.

```
7D00        LI R5,4
7D04        NOP
7D06        NOP
7D08        NOP
7D0A        BLWP @>6020
  .
  .
  .
```

Thanks to the free memory locations, you can correct the error, so that the program lines read:

```
7D00        AORG >7D04
7D04        CLR @>8374
7D06        END
```

Memory location >7D08 will still be free to add some other missing two-byte instruction if necessary.

Memory locations containing machine language translations of the NOP instruction are ignored by the *Assembler*. You should use NOP whenever you feel something might have to be added to the program later on.

It's wise to leave some NOP instructions when you are jumping to a label not yet defined with one of the jump instructions, because it may end up that when the label is finally defined, that it's beyond the 256-byte limit of the instruction. If there are no NOP instructions after or before the jump, you won't be able to change it to a B (Branch) instruction, because it uses two bytes more than the jump instructions.

The next example jumps to the label NT, which is beyond range when defined, causing an *R-ERROR*:

```
7D00        LI R2,5
7D04        CI R2,5
7D08        JEQ NT
```

```
7D0A        NOP
7D0C        NOP
7D0E   RN   (program continues)
   .
   .
   .
7F60   NT   CLR R2
7D08 *R-ERROR*
```

The *R-ERROR* message is caused by the jump (JEQ) instruction trying to transfer control to NT, which is too far away. The free locations left after the jump will let you correct the mistake, replacing the jump instruction with a branch instruction:

```
7F62        AORG >7D08
7D08        JNE RN
7D0A        B @NT
7D0E        END
```

The error has been corrected by inverting the jump, replacing it with a branch instruction, as you saw demonstrated in Chapter 3.

Copying Registers: MOV and MOVB Instructions

Many times you'll need to copy the value from one register to another register or memory location, or from a memory address to another address or register. In these cases, you'll need the MOV (MOVe word) and MOVB (MOVe Byte) instructions.

The MOV (MOVe word) instruction makes a copy of the word value in the source operand, placing it in the destination operand (refer to the short definition of source and destination operands in Chapter 3).

```
7D00        MOV R3,R4         (Place the word found in R3 in
                              R4, leaving R3 unchanged)
7D02        MOV R4,@>7E00     (Place the word found in R4 into
                              >7E00)
7D06        MOV @>7D24,R5     (Place the word found in location
                              >7D24 in R5)
```

The MOVB (MOVe Byte) instruction works in much the same way, but instead operates only with the left (most significant) bytes of the words.

```
7D00        MOVB R7,R2        (Copy the left byte of R7 into the
                              left byte of R2. The right byte of
                              R2 and the word in R7 remain
                              unchanged)
```

```
7D02     MOVB *R2+,@>7EF2 (Copy the left byte of the
                           word found in the address stored
                           in R2 and place it in the left byte
                           of the word in >7EF2. The value
                           in R2 is increased by one byte)
7D04     MOVB @>7EF2,*R8  (Copy the left byte of the word
                           in >7EF2 into the left byte of the
                           address stored in R8)
```

Saving Memory: Fewer Labels

Every time you use a label in your program, you're consuming four bytes of valuable memory. Though labels are convenient and easy to use, they should be avoided whenever possible.

When you create a label, it's added to the Symbol Table, which starts at location >7CD8. When you select the NEW option of the *Assembler*, you're placed at the default memory address of >7D00. Starting your program there will leave space in the Symbol Table for nine labels. If you use more than this, the table will overwrite the beginning of your program.

According to the number of labels you're using, you can decide where to start the program. Count the number of labels you're planning to use, and add one (because the computer adds a null entry as the last entry of the table). Multiply this number by four, because each entry in the Symbol Table occupies four bytes. Convert the answer to hexadecimal so you can add it to the location where the Symbol Table begins—this will give you the exact place in memory to start your assembly language program.

Assume your program will have 14 labels. The calculation to find the first free address for your program would be:

(14+1) × 4 = 60 (decimal)
60 = >3C (hexadecimal)
>7CD8 + >003C = >7D14

Location >7D14 is where you should start your program.

If your program will have only three labels, the calculation would be:

(3 + 1) × 4 = 16
16 = >10
>7CD8 + >0010 = >7CE8

So start your program at >7CE8.

Labels use up a lot of memory and can be avoided in several ways.

- Do not use the EQU directive. Instead of giving a memory location or routine a label, branch to it directly. The first example shows how the code would be written with a label; the second example illustrates avoiding a label.

```
1. 7D00   N3   EQU >6034
   7D00        BLWP @N3
2. 7D00   BLWP @>6034
```

- These two program segments have exactly the same result, but the second avoids using memory for the Symbol Table.
- If you know where you're going to place a subroutine, instead of branching to a label and assigning it later, branch directly to the starting memory address of the routine.

```
1. 7D00        BL @CC
    .
    .
    .
   7EF8   CC   CLR R0
2. 7D00        BL @>7EF8
    .
    .
    .
   7EF8        CLR R0
```

Again, the second example avoids using a label.

- The same method can be used with the jump and branch instructions:

Jump

```
1. 7D00        JMP Z5
    .
    .
    .
   7D50   Z5   LI R7,5
2. 7D00        JMP >7D50 (No @ is needed)
    .
    .
    .
   7D50        LI R7,5
```

Branch

```
1. 7D00        B @PQ
     .
     .
     .
   7E50   PQ   CLR R1
2. 7D00        B @>7E50
     .
     .
     .
   7E50        CLR R1
```

- You can also refer directly to a memory location when using a DATA table or text if you know where it is, or will be, in memory.

```
1. 7D00        LI R0,300
   7D04        LI R1,TX
   7D08        LI R2,5
   7D0C        BLWP @>6028
     .
     .
     .
   7F30   TX   TEXT 'LABEL'
2. 7D00        LI R0,300
   7D04        LI R1,>7F30
   7D08        LI R2,5
   7D0C        BLWP @>6028
     .
     .
     .
   7F30        TEXT 'LABEL'
```

Both these program segments display the word LABEL on the screen, but the second uses no label.

- The *Assembler* predefines the dollar symbol ($) to mean "the current memory location." This is a great help when your program is jumping around in memory without using labels. For instance, these two instructions mean the same thing.

```
1. 7D00        JMP $
2. 7D00   NQ   JMP NQ
```

The first example means to jump to the current memory location, which happens to be >7D00. The second, which also creates an endless loop, does the same, but it uses a label.

To jump three words (six bytes) forward in memory, you could write:

```
7D00        JMP $+6        (Control passes to >7D06, cal-
                           culated by adding six bytes to
                           >7D00)
```

To jump two words (four bytes) back in memory, you would subtract four bytes from the current memory location.

```
7D0A        JMP $-4        (Control passes to >7D06)
```

Executing Your Program

When you've finished writing an assembly language program, there are three ways to execute it. The first and most immediate method is to END the program, exit the *Assembler,* select EASY BUG and use the E (Execute) command, followed by the hexadecimal address where your program begins. This is the method you've used to execute the example programs so far.

The second way is to add the name and position of your program to the REF/DEF Table (Table of REFerences and DEFinitions) and execute the program like the LINES demonstration program, using the RUN option of the *Mini Memory* menu. (You'll see how to add the name and position to your program in just a moment.)

The third method also requires the name and starting position of the program added to the REF/DEF Table. To call the program from BASIC, you must use the CALL LINK subroutine with the following syntax:

CALL LINK (*"program name"*)

where *program name* is the name of the program as added in the REF/DEF Table. Care must be taken when linking BASIC and assembly language programs. Read Chapter 7 before trying to link your own programs.

To test the three execution methods just mentioned, load the LINES program from tape. To execute it from EASY BUG with the E (Execute) command, type:

? E7D9E (>7D9E is where LINES begins)

To run LINES from the *Mini Memory* menu, choose (2) RUN and when the PROGRAM NAME? message appears, type LINES and press ENTER.

To run LINES with the BASIC CALL LINK statement, select (1) TI BASIC from the module's title screen and then type, in immediate mode (in other words, without line numbers):

CALL LINK ("LINES") and press ENTER.

Adding Program Name and Position

Adding the name and position of your program to the REF/DEF Table, so that you can run it from *Mini Memory* or from BASIC, is relatively simple.

The REF/DEF Table starts at >7FFF and grows "backwards" toward >7000. That means it occupies the last portion of RAM memory in the module. Each entry is only eight bytes long—thus several program names can be added to the table. The program name uses six bytes of memory and the starting address uses two.

When you load the *Assembler*, the entry for the NEW option of the *Assembler* occupies addresses >7FF8 to >7FFF. The OLD option occupies addresses >7FF0 to >7FF7. LINES has its name and starting address from >7FEF to >7FE8.

You can add the name and position of your program *before* the entry for LINES (from >7FEF to >7FE8), or use the entry for LINES directly since your program will overwrite part or all of it anyway. Of course, your program must not be longer than the place where you will add the name and position of it in the table, or you'll overwrite your own program.

Two memory addresses tell you the First Free Address of the Module (FFAM) and the Last Free Address of the Module (LFAM). >701C contains the FFAM (first free address after your program is finished) and >701E holds the LFAM (the place in memory where your program name and position are loaded).

To check this, load LINES from tape and choose the NEW option of the *Assembler*. Then type:

```
7D00 045B          AORG >701C
```

to get to FFAM. You'll see:

```
701C 7FB2
```

This value (>7FB2) is the FFAM, the first free address after the program LINES is finished. Press ENTER to get to the next memory location, >701E, where LFAM is stored. Now you should see:

```
7D00 7FE8
```

This means that the last free position before the REF/DEF Table entry for LINES is >7FE7. The entry for LINES begins in >7FE8.

When you finish writing your program, you must update these values. Use AORG to get to the correct memory locations and DATA to place the correct values there. Remember that your program has to leave eight bytes for the name and starting address of your program. When you have updated the values in >701C and >701E, you can proceed to add your program entry to the REF/DEF Table.

The steps to change the FFAM and LFAM are:

- Use AORG to get to >701C, where the FFAM is stored.
- Use DATA to change the value there to the new first free address after your program is finished.
- Use DATA to change the value in >701E to the place in the REF/DEF Table where you'll add the entry for your program. To use the same entry as for LINES, use >7FE8. For the entry before LINES, use >7FE0. For each new entry, subtract eight bytes from the previous one. Check that the place where you'll make the entry for your program (the value you place in >701E) is at least seven or more bytes than the value in >701C, or there won't be a place to add the entry to your program.

Once you've updated the FFAM and the LFAM addresses in memory, you can add the name and starting position of your program to the REF/DEF Table. To do this, use the AORG Directive to get to the place in the table where you want to make the entry for your program. (This should be the same location you earlier stored in >701E.) Once there, use the TEXT directive to add the program name. The name can be one to six characters long—if it's shorter than six characters, you must pad the name with blanks. *The text you add with the TEXT directive must be six characters long.* Then use the DATA directive to add the starting position of your program into memory. If your program starts at >7D30, enter *DATA >7D30.* If the program's first instruction is labeled, with N5 for instance, you could instead use *DATA N5.*

An Example

It's always easier to understand something if you have an example in front of you. Let's do just that—we'll write a short routine which will print *TI-99/4A* on the screen, and then we'll save it in the REF/DEF Table under the name of *TI-99*.

```
7D00        LWPI >70B8         (Load memory area for the
                               registers)
7D04        LI R0,298          (Load screen printing position)
7D08        LI R1,TX           (Load position of text in memory)
7D0C        LI R2,8            (Length of text: eight bytes)
7D10        BLWP @>6028        (Print text on screen)
7D14        JMP $              (Create endless loop to stop the
                               program)
7D16   TX   TEXT 'TI-99/4A'    (Text to be displayed)

7D1E        AORG >701C         (Jump to the address with the
                               FFAM)
701C        DATA >7D1E         (Set the new FFAM)
701E        DATA >7FE0         (Set the new LFAM—where the
                               entry for the program will be
                               added to the REF/DEF Table)
7020        AORG >7FE0         (Jump to the location in the table
                               where you'll add the name and
                               position of the program)
7FE0        TEXT 'TI-99 '      (Program name: five characters
                               plus one blank)
7FE6        DATA >7D00         (Place in memory where the pro-
                               gram begins)
7FE8        END
```

Now run the program using the RUN option of the *Mini Memory* menu. You don't need to leave a blank space after *TI-99* when prompted for the program name.

This program will not work if it's called from BASIC. The reason is an existing screen bias discussed in Chapter 7.

More to Come

So far you've learned many of the basics of assembly language programming. You've been introduced to directives and instructions, and have even seen how to write and save a simple program. If you're at all confused about anything already covered, it would be best to go back and look over it again. We'll be exploring more and more complex techniques of assembly language programming as we continue.

In fact, the next chapter will show you how to create programs to read and control the keyboard and joystick. Almost all programs, from spreadsheets to arcade-quality games, use one or the other to get input from the user. In a short time you'll be able to design subroutines which allow assembly language speed in reading and using these input devices.

Chapter 5

Keyboard and Joysticks

Keyboard and Joysticks

All BASIC operations like INPUT, CALL KEY, ACCEPT AT, and CALL JOYST are executed in assembly language with the aid of the KSCAN (Keyboard SCAN) utility, which is stored at location >6020. This routine works in the same way as the CALL KEY subroutine in BASIC. To perform an INPUT or ACCEPT AT kind of operation in assembly language, we have to accept characters or numbers one at a time, checking to see if they are valid, printing them on the screen if they are, and storing them somewhere in memory until the code for the ENTER key is detected, indicating that the operation is complete. CALL KEY and CALL JOYST operations are also easy to write, as you'll see in a few moments.

Preparing for the KSCAN Routine

The KSCAN routine needs to know the keyboard device number when it's called, so this value has to be put into memory before the branch (BLWP) to the KSCAN utility is executed. The keyboard device number is the same as the key unit in the BASIC CALL KEY subroutine. A 0 means a standard keyboard scan, 1 is used to read joystick 1, and so on. This value has to be put in the byte at location >8374. To place a 0, just clear the memory address:

```
CLR @>8374
```

To place another value, load it into the left (most significant) byte of a register and then move that byte into the corresponding memory location:

```
LI R7,>0200
MOVB R7,@>8374
```

The above lines place a 2 (reading joystick number 2) into byte >8374.

Once this has been done, you can branch to the KSCAN routine in location >6020 with:

```
BLWP @>6020
```

If a key is pressed, its hexadecimal ASCII code is placed in byte >8375. You can detect whether a key was pressed by simply checking another byte, the *status byte*, at location >837C, just as is done with the *status* variable in the BASIC CALL KEY. In BASIC, if the status variable is 0, no key has been pressed. If it's −1, a key *has* been pressed. The status byte doesn't work quite like this.

Checking the Status Byte

Before we go on with anything else, we need to make a short sidetrip. A byte is divided into eight pieces, called *bits*. These eight bits, numbered 0 through 7 from left to right (the convention used by TI), may be either *set* (contain a 1) or *reset* (contain a 0). The following byte has bits 2 and 4 set and the rest reset:

0	1	2	3	4	5	6	7	**Bit number**
0	0	1	0	1	0	0	0	**Condition (set or reset)**

The values of each set bit double as you move from right to left. Bit 7, the bit on the far right, has a value of 1 when it's set. Bit 6 has a value of 2, bit 5 has a value of 4, and so on, until bit 0 has a value of 128. It looks like this:

0	1	2	3	4	5	6	7	**Bit number**
128	64	32	16	8	4	2	1	**Bit value when set**

To get the total value of a byte, simply add together the values of the bits which are set. The byte above, then, would have a total value of 40 (32+8).

What makes the status byte so important for the KSCAN routine is that bit number 2 is set if a key *is* pressed. Otherwise, it's reset. The other bits in that byte don't interest us. By checking the condition of the second bit in the status byte, you can know whether a key was pressed. To do this checking, you'll use the COC (Compare Ones Corresponding) instruction. This instruction compares the bits set in the first operand to the bits set in the second operand. If all bits set to 1 in the *first* operand have a corresponding bit set to 1 in the *second* operand, the operands are considered equal. It doesn't have to be reciprocal. For example, these two bytes are considered equal by the COC instruction:

First operand byte	1 0 1 0 0 1 0 0
Second operand byte	1 1 1 0 1 1 0 1

All the bits set in the first byte have a corresponding bit set in the second byte. If the bytes were reversed (the first becomes the second, the second the first):

First operand byte	1 1 1 0 1 1 0 1
Second operand byte	1 0 1 0 0 1 0 0

the bytes would be considered unequal by the COC instruction because not all the set bits in the first byte have a corresponding set bit in the second byte.

So how can all this help? If you create a byte with only the second bit set, like this

0 0 1 0 0 0 0 0

and compare it to the status byte with the COC instruction, the bytes will be considered equal if the second bit of the status byte is set and different if it is reset, regardless of the other bits. Then you'll know that if the two compared bytes are equal, a key *was* pressed. If they're different, no key was pressed.

This instruction places a byte with only bit 2 set in the left byte of the memory location labeled MR:

MR DATA >2000

>20 in hexadecimal is 32 in decimal—that's the binary number *00100000*—the value needed. The right byte of location MR was filled with zeros so as not to affect the comparison.

Before we go on to see an example, be assured that nothing is amiss if this last section has left you a bit (no pun intended) confused. As you start getting used to thinking in assembly language, you'll understand these explanations much better. For the time being, it's enough to just understand the *method* to perform a KSCAN, not really about why it's done a certain way.

The labeled memory location with the comparison byte is usually placed at the end of the program where it won't be executed as an instruction. Then, when you check whether a key was pressed in the KSCAN loop, you can clear a register (to make sure the unused byte of it is set to zero), move the status byte into the register, and compare it to the labeled value. Like this:

```
CLR R1             (Clear R1 to receive status byte)
MOVB @>837C,R1     (Move status byte into left byte of R1)
COC @MR,R1         (Compare bits set to 1 of the value stored at
                   MR to the bits set to 1 in R1)
```

Then, if the result of the COC indicates that the operands are equal, a key *was* pressed. Otherwise, no key was pressed.

Displaying a Message

The next example demonstrates the technique to perform a KSCAN, checking the status byte with the COC instruction. Note the way that the KSCAN operation is written. The following program waits for you to press a key and then displays the message KEY DETECTED on the screen:

Key Detector

```
7D00      LWPI >70B8         (Load memory area for registers)
7D04      CLR @>8374         (Clear byte >8374: standard key-
                             board scan)
7D08  LP  BLWP @>6020        (Branch to the scanning routine)
7D0C      MOVB @>837C,R1     (Move status byte into R1)
7D10      COC @BT,R1         (Compare set bits of the compari-
                             son value added at the end of the
                             program, with label BT, to the set
                             bits of the word in R1)
7D14      JNE LP             (The second bit of R1 is not set
                             and the bytes are considered dif-
                             ferent by the COC instruction: No
                             key was pressed, so return to loop
                             LP)
7D16      LT R0,298          (Key was pressed. Load screen po-
                             sition to display the message)
7D1A      LI R1,TX           (Load position of text in memory)
7D1E      LI R2,12           (Load length of text)
7D22      BLWP @>6028        (Display text on the screen)
7D26      B *R11             (Return to EASY BUG)
7D28  BT  DATA >2000         (Comparison value for the
                             KSCAN loop)
7D2A  TX  TEXT 'KEY DETECTED' (Text to be displayed)
7D36      END
```

Run the program. The message will be printed on the screen the moment you press any key.

Computer Typewriter

This next example takes further advantage of the KSCAN routine by reading the ASCII value of the key pressed and printing the corresponding character on the screen, just as if the computer were a typewriter. (Certainly not a word processor, but this is an important part of any word processing program.)

The first part of "Computer Typewriter," with the keyboard reading loop, is the same as "Key Detector," the previous program. However, instead of printing a message when a key is pressed, Computer Typewriter reads the ASCII of the key pressed and displays the corresponding character on the screen, returning to the KSCAN loop for a new keypress.

Computer Typewriter

```
7D00       LWPI >70B8        (Load memory area for the
                             registers)
7D04       CLR R0            (Initial screen printing position)
7D06       CLR @>8374        (Clear byte >8374. Standard key-
                             board scan)
7D0A  LP   BLWP @>6020       (Branch to KSCAN routine)
7D0E       MOVB @>837C,R1    (Move the status byte into R1)
7D12       COC @BT,R1        (Compare both bytes with the
                             COC instruction)
7D16       JNE LP            (Bytes different, no key pressed,
                             return to loop. Else continue
                             program)
7D18       MOV @>8374,R1     (Key was pressed. Move the byte
                             with the key's ASCII value [in
                             >8375] into the least significant
                             byte of R1 by moving the com-
                             plete word at >8374 over)
7D1C       SWPB R1           (For the VSBW routine, the key's
                             ASCII value must be in the left
                             byte of R1, so switch bytes)
7D1E       BLWP @>6024       (Write the character to the screen)
7D22       INC R0            (Increase printing position)
7D24       CI R0,768         (Has the last screen position been
                             passed?)
7D28       JLT LP            (No. Return to loop for a new key)
7D2A       CLR R0            (Yes. Reset screen value)
7D2C       JMP LP            (Return to loop for a new key)
7D2E  BT   DATA >2000        (Comparison value for the
                             KSCAN loop)
7D30       END
```

When you run this program, each key you press prints on the screen. Some keys with no printable codes (such as the EN-TER key) will just print as blanks. If the last screen position is reached, the new characters will begin to overprint the old ones. Here are some characters you might like to see printed (executing with EASY BUG):

FCTN - 1
FCTN - 2
FCTN - 3
FCTN - 4
FCTN - 7 (Parts of the TI title screen map)
FCTN - 8
FCTN - S
FCTN - D
FCTN - X (The copyright symbol from the computer's title screens)

Note that the character is not printed again if you keep on pressing the key. This might be a problem with some kinds of scans, as you'll see later on.

An Assembly Language INPUT

To execute an INPUT operation in assembly language, you must print each character read and accepted by the KSCAN loop on the screen, store it in some memory area, and check to see whether ENTER was pressed to end the input. If it was, the KSCAN loop ends. If not, the program returns to the loop for a new key. You should keep track of how many characters there are in the input value or string.

The next program simulates BASIC's INPUT (without the beep sound and flashing cursor), letting you input a string of characters with no restrictions. It then prints the string beneath whatever you entered. The loop to read the keyboard is the same as the previous examples. Each time a key is pressed, its hexadecimal ASCII code is moved into a register to be printed on the screen; it's also moved to a memory location for storage. When the ENTER code is detected, 13 (>0D), the program ends the KSCAN loop and prints whatever you typed in. The assembly language routine will be similar to the BASIC statements:

```
100 INPUT A$
110 PRINT A$
```

Assembly Language INPUT

7D00	**LWPI >70B8**	(Load memory area for registers)
7D04	**CLR @>8374**	(Clear byte >8374. Standard keyboard scan)

```
7D08        LI R0,100          (Load screen position to print the
                               prompt)
7D0C        LI R1,>3F00        (Load code for the question mark
                               prompt)
7D10        BLWP @>6024        (Print the question mark)
7D14        INCT R0            (Prepare the screen position to re-
                               ceive input)
7D16        CLR R7             (Prepare a register to store the
                               number of characters in input)
7D18        LI R9,>7F00        (Load R9 with the memory area
                               where the input will be stored)
7D1C  LP    BLWP @>6020        (Start the KSCAN loop to read the
                               keyboard)
7D20        MOVB @>837C,R1     (Move status byte into R1)
7D24        COC @>BT,R1        (Compare Ones Corresponding to
                               the value in BT)
7D28        JNE LP             (If the values are different, no key
                               has been pressed so return to
                               loop)
7D2A        MOV @>8374,R1      (Move the key pressed into the
                               least significant byte of R1. The
                               left byte is zero because byte
                               >8374 had been cleared)
7D2E        CI R1,13           (Was ENTER key pressed?)
7D32        JEQ CT             (Yes. Jump to CT to continue the
                               program)
7D34        SWPB R1            (No. Place the keycode in the left
                               byte of R1)
7D36        INC R7             (Input has one more character. In-
                               crease character counter in R7)
7D38        BLWP @>6024        (Print the character on the screen)
7D3C        MOVB R1,*R9+       (Store ASCII in memory address
                               found in R9 and increment this
                               address by one byte, for the next
                               character code to be stored)
7D3E        INC R0             (Increase screen printing position)
7D40        JMP LP             (Return to KSCAN loop for a new
                               key)
7D42  CT    CI R7,0            (Start routine when ENTER is
                               pressed. Check that there is at
                               least one character to be printed)
```

```
7D46        JEQ LP          (No character in user's input. Do
                            not accept the ENTER key and re-
                            turn to the KSCAN loop)
7D48        LI R0,164       (Load R0 with the screen position
                            to print the input)
7D4C        LI R1,>7F00     (Place in memory where the input
                            is found, is loaded into R1)
7D50        MOV R7,R2       (Load R2 with the length of the
                            text. This is the value kept track of
                            in R7)
7D52        BLWP @>6028     (Print input on the screen)
7D56        JMP $           (Endless loop)
7D58   BT   DATA >2000      (Comparison value for the
                            KSCAN loop)
7D5A        END
```

It's necessary to check that the input is at least one character long, for the VMBW routine won't work if the number of bytes to be displayed is zero. Since the program keeps track of the string length in R7, it can also check whether the value stored there is zero (>7D42). If it is, the program returns to the KSCAN loop, ignoring the ENTER keypress.

Limiting the length of the input is also helpful when displaying the string of characters on the screen with the VMBW utility. If this value was fixed, and the input was longer, it would be truncated and the remaining characters ignored. If the input was shorter than the number of bytes in R2, the computer would just read the values in the following memory locations, whatever they were, and print them on the screen after the actual input.

When you run the program, a question mark prompt appears at the top of the screen. Type in whatever you want and press ENTER. Whatever you entered appears below the input.

Repeating Keys

The previous examples are useful when you have to perform an INPUT operation in assembly language, or in cases when you want a key to be read only once each time it's pressed. In other words, programs where you want a key to be detected continuously, not having to release the key first and then press it again, will not work with the previous method.

We'll look at a routine which produces a repeating keystroke shortly. First, however, let's see the nonrepeating version of this routine.

It moves the plus sign (+) around the screen, leaving a trail. You guide the + sign with the arrow keys. No screen limit checks are executed, so don't move the + off the top or bottom of the screen, or you'll be writing into other memory areas and the program might crash. This example uses the status byte checking method to read the keyboard, so you'll have to press and release a key each time you want the + to move one space in any direction.

Moving Plus—Single Keystroke

```
7D00        LWPI >70B8          (Load memory area for registers)
7D04        LI R0,300           (Initial screen printing position)
7D08        CLR @>8374          (Clear byte >8374. Standard key-
                                board scan)
7D0C   LP   BLWP @>6020         (Branch to KSCAN routine)
7D10        MOVB @>837C,R1      (Move status byte into R1)
7D14        COC @BT,R1          (Use the COC instruction to com-
                                pare the set bits of the value in BT
                                to the word in R1)
7D18        JNE LP              (If the bytes are different, return
                                to loop)
7D1A        MOV @>8374,R1       (Move the ASCII code of the key
                                pressed into the right byte of R1)
7D1E        CI R1,68            (Was the D key pressed?)
7D22        JNE $+6             (No. Jump six bytes forward)
7D24        INC R0              (Yes. Move printing position one
                                right, since right arrow [D] was
                                pressed)
7D26        JMP PR              (Printing position updated. Go to
                                the printing routine)
7D28        CI R1,83            (Was the S key pressed?)
7D2C        JNE $+6             (No. Jump six bytes forward, to
                                the next comparison routine)
7D2E        DEC R0              (Key pressed was the left arrow
                                [S]. Move printing position left
                                one by decreasing the value in
                                R0)
7D30        JMP PR              (Position updated. Jump to print-
                                ing routine)
```

```
7D32        CI R1,69          (Was the E key pressed?)
7D36        JNE $+8           (If not, jump four words forward)
7D38        AI R0,-32         (Up arrow [E] pressed. Move the
                              + one position up)
7D3C        JMP PR            (Jump to the + printing routine)
7D3E        CI R1,88          (Was the X key pressed?)
7D42        JNE LP            (If not, jump back to the KSCAN
                              loop)
7D44        AI R0,32          (Down arrow [X] pressed. Move
                              printing position one down by
                              adding 32 to the current position)
7D48   PR   LI R1,>2B00       (Printing routine. Load R1 with
                              the code for the +)
7D4C        BLWP @>6024       (Print the + sign)
7D50        JMP LP            (Jump back to the loop for a new
                              key)
7D52   BT   DATA >2000        (Comparison value for KSCAN
                              loop)
7D54        END
```

End the program and execute it. The + sign appears when you press a key. Try it out—move the symbol around the screen with the arrow keys.

But this program would be much more efficient if the symbol would continue moving as long as a key was held down. Another KSCAN method can be used to do that.

When you want a key to be read continuously, you can directly check the ASCII value of the key in location >8375 and branch accordingly. If no key was pressed, the value in >8375 will be >FF. Study the following program, which does the same thing as the previous example—the only real difference is that it reads the value in location >8375 to perform the KSCAN loop. (Since this is so much like the previous program, it's not commented, with one exception.)

Moving Plus—Repeating Keys

```
7D00        LWPI >70B8
7D04        CLR @>8374
7D08        LI R0,300
7D0C   LP   BLWP @>6020
7D10        MOV @>8374,R1     (Move ASCII of key pressed to the
                              right byte of R1)
```

```
7D14        CI R1,68
7D18        JNE $+6
7D1A        INC R0
7D1C        JMP DR
7D1E        CI R1,83
7D22        JNE $+6
7D24        DEC R0
7D26        JMP DR
7D28        CI R1,88
7D2C        JNE $+8
7D2E        AI R0,32
7D32        JMP DR
7D34        CI R1,69
7D38        JNE LP
7D3A        AI R0,-32
7D3E   DR   LI R1,>2B00
7D42        BLWP @>6024
7D46        JMP LP
7D48        END
```

End the program and run it. Be *very* careful when you press the arrow keys (hardly touch them). The program has no delays and the + sign might move a little faster than expected. That's part of the magic of assembly language programs—things move quickly, far more quickly than in BASIC.

No screen limit check is made, so start off by pressing the left or right arrow keys so the + will not move into other memory areas. If you wish, you can add a delay loop after the + symbol is printed to slow down program execution.

Two Joystick Reading Routines

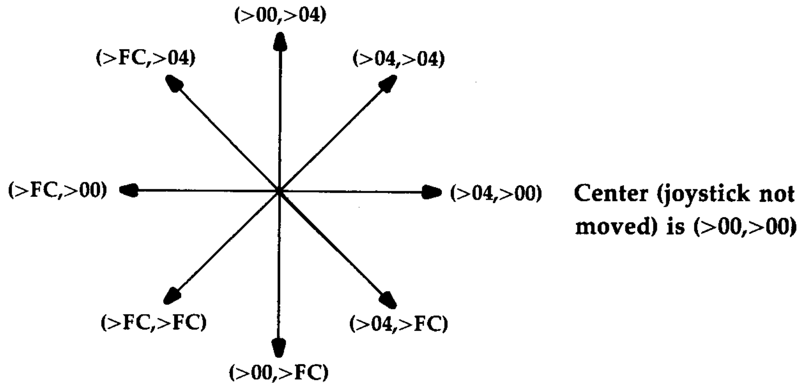
Now that you know how to read the keyboard for input, whether it's single keypresses or repeating keys, you have one of the most important elements of programming available. After all, what program doesn't take some sort of input from the user via the keyboard?

How about games? Many games (and even programs that aren't games) don't use the keyboard, or if they do, use it infrequently. Instead, most games use a joystick for user input. How can you read and use joysticks in assembly language?

It's not any more complex than what you've already done. Joysticks 1 and 2 are read with the KSCAN routine, just as the keyboard is.

To read joystick 1, you must place a 1 in the byte at location >8374. To read joystick 2, load a 2 in the same byte. When you branch to the KSCAN routine, the *Y-return* of the joystick (see the *User's Reference Guide* for details of Y-return and its companion, X-return) is placed in location >8376 and the *X-return* is placed in location >8377. By checking these values, you can tell in what direction the joystick was moved. The figure below illustrates the hexadecimal values placed in both bytes when the joystick is moved in a particular direction. The first number indicates the value placed in >8377 (X-return) and the second number the value in >8376 (Y-return).

Joystick Values



When the joystick is pushed forwards (to the north), the value in >8377 would be >00 and the value in >8376 >04. If the joystick was left centered, the value in both bytes would be >00. The other directions produce their appropriate values in these two memory addresses.

Making comparisons for all eight possible directions can be a tedious task. Oftentimes, programmers, even those who use assembly language, use only four positions of the joystick. The top two diagonals can be considered as up or north movements, while the bottom diagonals can be considered as down or south. The two left diagonals can indicate a left or west movement; the two diagonals on the right would then be right or east movements. The necessary comparisons are then reduced to four.

In the first joystick reader example, we'll read just four directions. In the second sample, however, all eight directions will be read so you can see how it's done.

Moving Cross

The following example moves an X around the screen in the direction you move the joystick. There are no screen limit checks, so don't move the X off the top or bottom of the screen.

```
7D00        LWPI >70B8         (Load memory area for registers)
7D04        LI R0,300          (Initial screen printing position)
7D08        LI R1,>0100        (Load joystick number in left byte
                               of R1)
7D0C        MOVB R1,@>8374     (Move joystick number into byte
                               >8374, so joystick 1 will be read
                               by the KSCAN loop)
7D10   LP   BLWP @>6020        (Branch to the KSCAN routine)
7D14        CLR R1             (Clear register where the value in
                               >8376 will be placed. It's cleared
                               to insure that its right byte will be
                               zero in the comparison).
7D16        MOVB @>8376,R1     (Move the value in >8376 [Y-re-
                               turn] into the left byte of R1)
7D1A        CI R1,>0400        (Is the Y-return >04, meaning the
                               joystick was moved in one of the
                               three up directions?)
7D1E        JNE T1             (If it was not, jump to the second
                               comparison, starting at T1)
7D20        AI R0,-32          (Joystick moved in one of the up
                               directions. Move printing position
                               one up by subtracting 32 from the
                               current position)
7D24        JMP PG             (Position updated. Jump to print-
                               ing routine)
7D26   T1   CI R1,>FC00        (Second comparison. Was joystick
                               moved in one of the down direc-
                               tions—is Y-return >FC?)
7D2A        JNE T2             (If it wasn't, jump to third
                               comparison at T2)
7D2C        AI R0,32           (Move printing position one down
                               by adding 32 to the current
                               position)
7D30        JMP PG             (Position updated. Jump to print-
                               ing routine)
7D32   T2   MOVB @>8377,R1     (Third comparison. Check the X-
                               return, so move the byte in >8377
                               into the left byte of R1)
```

```
7D36        CI R1,>0400     (Is the X-return >04, meaning the
                            joystick was moved right?)
7D3A        JNE T3          (If not, jump to last comparison at
                            T3)
7D3C        INC R0          (Update printing position by add-
                            ing one to current position)
7D3E        JMP PG          (Jump to printing routine)
7D40   T3   CI R1,>FC00     (Last comparison. Was joystick
                            moved left?)
7D44        JNE LP          (If not, return to KSCAN loop be-
                            cause the joystick remained
                            centered)
7D46        DEC R0          (Update printing position by add-
                            ing one to current position)
7D48   PG   LI R4,3000      (Printing routine. Start delay loop)
7D4C        DEC R4          (Decrease the value in R4 by one)
7D4E        JNE $-2         (If it is not zero, delay loop still
                            incomplete. Jump back one word
                            in memory to the DEC
                            instruction)
7D50        LI R1,>5800     (Load ASCII code for the X in R1)
7D54        BLWP @>6024     (Print the X on the screen)
7D58        JMP LP          (Return to loop to read the joy-
                            stick once again)
7D5A        END
```

A delay loop was added to the program to slow down execution. For the program to run at different speeds, change the value stored in R4, which is loaded in memory address >7D48. (Choose the OLD option of the *Assembler*, and do not rewrite the line with the label PG; simply enter *7D48 LI R4,XXX*, where *XXX* is the new delay you want to use.)

The KSCAN loop is prepared by loading location >8374 with the joystick number (1 in our example). Then the branch to the KSCAN utility is executed and the value in >8376 (Y-return) is moved into R1 for checking purposes. If the value is >04, the joystick was moved up or in one of the up diagonals—thus the X character is moved up one screen line by subtracting 32 from its position. If the value returned is >FC, the joystick was moved in one of the three down directions—the X is moved one screen line down by adding 32. If the Y-return was none of these values, the X-return is checked in location >8377.

If it's equal to >FC, the X character is moved one position left. If it's >04, the X moves one position right. Then the program returns to the main loop to read the joystick once again.

Moving Cross—Reading Diagonals

This program works practically the same as the previous one—the major exception is that diagonals are also read. The only addresses commented are those which are additions to the previous program. The other instructions can be followed quite simply by referring to the "Moving Cross" program and its explanations.

```
7D00        LWPI >70B8
7D04        LI R0,300
7D08        LI R1,>0100
7D0C        MOVB R1,@>8374
7D10   LP   BLWP @>6020
7D14        CLR R1
7D16        MOVB @>8376,R1
7D1A        CI R1,>0400
7D1E        JNE C1
7D20        MOVB @>8377,R1      (Joystick moved in one of the
                                up directions. Check the X-
                                return to know which one)
7D24        CI R1,0             (Is the X-return equal to zero?)
7D28        JNE $+8             (If not, jump eight bytes
                                forward)
7D2A        AI R0,-32
7D2E        JMP DR
7D30        CI R1,>0400         (Was joystick moved diagonally
                                up and right?)
7D34        JNE $+8             (If not, jump eight bytes
                                forward)
7D36        AI R0,-31           (Update printing position to
                                move the X up one line, then to
                                the right by one position)
7D3A        JMP DR
7D3C        AI R0,-33           (The only possible position left
                                for the joystick to have been
                                moved is up and left, so update
                                the position to move one line
                                up and one position left)
```

```
7D40        JMP DR
7D42   C1   CI R1,>FC00          (Was joystick moved in one of
                                 the down directions?)
7D46        JNE C2               (If not, jump to the last
                                 comparison)
7D48        MOVB @>8377,R1       (Yes. Check the X-return to
                                 know in what down direction
                                 the joystick was moved)
7D4C        CI R1,0              (Was the joystick moved
                                 straight down?)
7D50        JNE $+8              (If not, jump eight bytes
                                 forward)
7D52        AI R0,32             (It was. Add 32 to the current
                                 position, to move one screen
                                 line down)
7D56        JMP DR
7D58        CI R1,>FC00          (Was lever moved down and
                                 left?)
7D5C        JNE $+8              (If not, jump forward eight
                                 bytes)
7D5E        AI R0,31             (It was. Update position by
                                 adding 31. Screen position will
                                 be moved one line down and
                                 one space left)
7D62        JMP DR
7D64        AI R0,33             (If lever was not moved down
                                 or down and left, then it was
                                 moved down and right. Update
                                 printing position by adding 33
                                 to the current position)
7D68        JMP DR
7D6A   C2   MOVB @>8377,R1       (If lever was not moved in any
                                 of the up or down directions,
                                 then it was either moved west
                                 or east [left or right]. Move X-
                                 return into R1 to check this)
7D6E        CI R1,>FC00          (Was lever moved left?)
7D72        JNE $+6              (If not, jump six bytes forward)
7D74        DEC R0               (It was. Decrease printing po-
                                 sition by one)
7D76        JNE LP
7D7E        INC R0               (Lever moved right. Update
                                 position)
```

```
7D80   DR  LI R1,>5800
7D84       BLWP @>6024
7D88       LI R4,3000
7D8C       DEC R4
7D8E       JNE $-2
7D90       JMP LP
7D92       END
```

End the program and run it. If the X character doesn't move diagonally, it's probably because you have a stiff joystick. Many joysticks, primarily the newer ones from TI, seem to have trouble detecting the diagonals.

Note that when a diagonal movement is detected in the program, the position is adjusted by adding or subtracting a value which changes the position one line, then one character left or right. Also note that R1 is cleared before a value is loaded into it. This is so that we can be sure that the right byte of R1 is 0. Many times when you're comparing two bytes with a word instruction, the bytes might actually be equal but considered different because the right bytes are not the same. Always try to make sure that the right bytes will be equivalent, generally by zeroing out the unused bytes of the word.

Chapter 6

Utilities, Mathematics, and Scrolling

Utilities, Mathematics, and Scrolling

Utilities

You've just seen how to write some of your own assembly language routines on the TI-99/4A. There are others, however, built-in routines, that are available to you. These ROM routines can save you considerable time and effort, for they're already in your computer. You don't have to sit down and write them. Instead, you can access them directly through assembly language.

This chapter shows you how to use two of the three utility routines, called the Extended Utilities, which in turn call other routines stored in your TI's ROM and GROM. The three routines are: GPLLNK (LiNK to GPL routines in GROM), XMLLNK (LiNK to routines in ROM), and DSRLNK (LiNK to Device Service Routines). We'll just look at the first two for now.

You have to be careful when linking to a preprogrammed routine in ROM or GROM. Make sure you've loaded the correct values in the correct addresses before calling the routine. Just as important, make sure that it doesn't destroy any memory areas where you've stored values.

Using GROM routines. To branch to a routine in GROM, use the GPLLNK utility, which is located at memory address >6018. To execute it, enter:

```
BLWP @>6018
```

The GROM routine you want to execute must be specified with a DATA statement following the call to the subroutine in GROM. For example, to call the accept tone GROM routine (which is routine number 34), you would type:

```
BLWP @>6018
DATA >0034
```

When a program has to execute sounds, allow automatic sprite motion, enable the FCTN = (QUIT) key, and so on, you have to allow *program interrupts.* This means that the program will quickly check whether any operation has to be executed, and if so, do it. It's best to quickly enable interrupts and disable them again with the LIMI instruction (Load Interrupt

Mask Immediate), as it's dangerous to allow a program to continue running with the interrupts enabled. That's because if you access VDP RAM while they are enabled, other values in VDP will be changed, causing strange effects in your program. VDP interrupts are disabled by default, so you will first have to enable them with the instruction *LIMI 2,* and then disable them again before accessing VDP RAM with the instruction *LIMI 0.* If there is a program segment constantly executed in your program (most programs have at least one), you can quickly enable and disable the program interrupts there. That will be sufficient for the computer to execute any operation which requires having VDP interrupts enabled. Just add these two lines in the frequently executed program segment:

```
LIMI 2
LIMI 0
  .
  .
  .
```

The computer will come across these two instructions, quickly enable and disable the interrupts, and then continue. In this way, you don't need to worry whether the interrupts are enabled or disabled at some obscure point in your program.

GROM routine access. The following program issues an accept tone when executed. The program will not work when run with EASY BUG's EXECUTE command, so the name and position will be added to the REF/DEF Table. Do this whenever your program will be using routines in GROM and/or ROM. No special data setup is necessary to call the routine.

Accept Tone

```
7D00        LWPI >70B8       (Load memory area for registers)
7D04        BLWP @>6018      (Link to GROM routine)
7D08        DATA >0034       (Number corresponding to the
                             GROM routine)
7D0A   LP   LIMI 2           (Start loop and allow VDP inter-
                             rupts for the sound to be
                             generated)
7D0E        LIMI 0           (Disable VDP interrupts)
7D12        JMP LP           (Stay in the loop to stop program
                             execution)
7D14        AORG >701E
```

```
701E        DATA >7FE0
7020        AORG >7FE0
7FE0        TEXT 'BEEP  '     (Add name and position of the
                              program to the REF/DEF Table.
                              Two spaces after the four-
                              character name)
7FE6        DATA >7D00
7FE8        END
```

If you change the value of the DATA statement in location >7D08 to >0036, a bad value sound will be generated.

Another example is the routine to execute the power-up operation. The following program executes that routine, causing the same effect as pressing FCTN = (QUIT) during the execution of a BASIC program. As soon as you run the next program, the computer will reset itself:

FCTN = (QUIT)

```
7D00        BLWP @>6018       (Execute GROM routine)
7D04        DATA >0020        (Routine to be executed is power-
                              up routine)
7D06        END               (End program—after the power-
                              up routine is executed, program
                              execution stops, so it's not nec-
                              essary to stop program with an
                              endless loop)
```

End this program and execute it. Remember that the name and the position of the program had to be added to the REF/DEF Table, since the program won't run from EASY BUG. If you've entered the Accept Tone program, just use the same name to run this new routine—the entry point for both is location >7D00.

Executing ROM routines. ROM routines can be executed in the same way as GROM routines. By using the XMLLNK utility, you can access a ROM routine. All you have to do is branch to >601C and specify the desired routine with a DATA statement.

```
BLWP @>601C       (Branch to ROM routine)
DATA >1200        (Convert floating-point number to integer
                  number)
```

Remember that some memory addresses may be overwritten when you use a ROM routine. Double-check that those addresses contain no data necessary to your program. If the data is needed, move it to some other area. Make sure to load the correct data into the correct memory addresses.

You have to be quite careful when using mathematical routines, both in ROM and GROM, as they work with floating-point values and not with integer values. There *is* a routine which resides in ROM, routine number >2300, which converts an integer value to a floating-point value, but it can't be used in your program unless you have both the memory expansion and the *Editor/Assembler*. You can find more details of this in Chapter 7.

Using routines in ROM and GROM requires some understanding of assembly language, so it may be a good idea to avoid them until you have a firm basis in assembly language programming. If you're just beginning to use assembly language, you can always come back to these routines later.

Mathematics in Assembly Language

How about two more assembly language instructions? You'll undoubtedly find uses for these instructions, for they perform multiplication (MPY) and division (DIV).

Multiplying two numbers. To multiply two numbers in assembly language, you can use the MPY (MultiPlY) instruction. This instruction uses two operands; the first may be a register, a memory address, or a label representing a memory address, while the second must be a register. Once the multiplication is executed, the answer is placed in the second operand (the workspace register) *and* the next workspace register. For example:

```
MPY R3,R4
```

multiplies the value in R3 by the value in R4 and places the answer in R4 *and* R5. If the answer is smaller than >FFFF, it fits entirely in R5. Otherwise, it uses both R4 *and* R5 (that's why two registers are used).

Let's suppose R3 contains a 1 (>0001) and R7 contains a 7 (>0007). If you then enter:

```
MPY R3,R7
```

the answer (>0007) is placed in R7 and R8. Because >0007 is less than >FFFF, the maximum value which can be represented by a memory word, it fits entirely in R8. The value in

R8 is now >0007 and the value in R7 is >0000. R3 remains unchanged.

The first operand may also be a memory address:

MPY @>7FA0,R4	(Multiplies the value at location >7FA0 by the value in R4, and places the answer in R4 and R5)

Or it can be a memory address with an assigned label:

MPY @NM,R8	(Multiplies the value stored at the address with label NM by the value in R8, then places the answer in R8 and R9)

Dividing values. The DIV (DIVide) instruction works much like MPY. The second operand (two memory words) is divided by the first operand. The integer result is placed in the *first word* of the second operand, the remainder in the second word of that same second operand.

DIV R3,R7

for instance, divides the value in R7 and R8 by the value in R3, and then places the integer result in R7. If there's any remainder, it's put in R8.

If the value you're going to divide (the second operand) can be represented by just one memory word, and is stored in R4, for example, don't use R4 in the DIV instruction. Instead, use R3. You can do this because the second operand is a two-word memory area (in this case using both registers 3 and 4). Assume that R3 is loaded with a four and R9 with a nine. R8 must contain a zero so that the value represented by R8 and R9 can be nine. Then the instruction

DIV R3,R8

divides the contents in R8 *and* R9 (>00000009) by the contents in R3 (>0004). The answer is placed in R8 (>0002) with the remainder in R9 (>0001).

You can also divide a value in a pair of registers by the value at a memory address.

DIV @>7B74,R7

divides the value in registers R7 and R8 by the value stored at location >7B74. The integer result is placed in R7 and the remainder in R8.

Register Shifting

In some cases you can use an easier method to multiply or divide. If you want to multiply a value by 2, 4, 8, 16, 32, and so on, and the result of the multiplication will be less than >FFFF (65535 in decimal), or if you want to divide a number by 2, 4, 8, 26, 32, you can shift the register to arrive at an integer result (with no remainders). The register shift instructions are Format V instructions. There are four of them: SLA (Shift Left Arithmetic), SRA (Shift Right Arithmetic), SRC (Shift Right Circular), and SRL (Shift Right Logical).

We'll be looking at just two of these instructions for now, SLA and SRL.

SLA (Shift Left Arithmetic). This instruction moves the bits in a word a determined number of positions left, filling the vacant positions with zeros. Imagine the following word (two bytes with eight bits each) as the content of workspace register 3:

```
1000101011101110
```

If you used the following instruction

```
SLR R3,3
```

each bit in the word in R3 would be moved three positions to the left. Vacant positions are set to zero, so the word in R3 would now be:

```
0101011101110000
```

This is useful, because, as you might have realized if you're familiar with binary, moving each digit of the number one column to the left is the same as multiplying the binary number by two. Moving each digit two positions to the left is like multiplying the number by four, moving each three places is the same as multiplying by eight, and so on.

In other words, if you have the binary number 00000101, which is 5 in decimal (the bit on the far right has a value of 1, the bit to its left has a value of 2, the next bit to the left has a value of 4, and so on, until the last bit has a value of 128) and you shift each digit two positions to the left (giving you 00010100), it's the same as multiplying 5 by 4. The result is 20, the same decimal value as 00010100.

The following instruction multiplies the value in R7 by 16:

SLA R7,4

and leaves the answer stored in R7.

Register shifting can also be used for other applications, such as moving the least significant byte of a word to the position of the most significant byte. This is done simply by shifting the value in the register eight bits to the left. The difference with the SWPB instruction is that the other byte is set to zero. For example:

```
1. LI R7,>B8A5
   SLA R7,8
2. LI R7,>B8A5
   SWPB R7
```

The first program segment leaves R7 loaded with >A500, while the second leaves R7 loaded with >A5B8. For some applications, the first method will prove more efficient.

SRL (Shift Right Logical). The SRL instruction shifts the bit of a word a number of positions to the right, filling the vacant places with zeros. The opposite of SLA, you can use this instruction to divide a value by 2, 4, 8, 16, and so on. If the answer to the division is not exact, you'll receive as answer the integer value of the floating-point result. If you need to know the remainder, you'll have to use the DIV instruction to perform the operation.

If R7 is loaded with 32 (00100000), the instruction

SRL R7,3

shifts the bits of the word in R7 three positions to the right, thus dividing the value in R7 by 8. The integer answer remains in R7, so the number would now be 00000100 (or 4 in decimal).

Use these two instructions to multiply and divide whenever you can—it can greatly simplify your programming work.

Finding the Absolute Value

In many applications, you might need to find the absolute value of a number. In those cases, the ABS (ABSolute value) instruction can be used. If a register is loaded with −32, and the ABS instruction is used, the value in that register is changed to 32. Notice that the only change is that the negative number becomes positive. However, if the value in the register is positive, say 32, then it would be left unchanged.

The number can be loaded in a register or at a memory address before its absolute value is calculated.

ABS R7	(Computes the absolute value of the word loaded in R7)
ABS @>7E00	(Computes the absolute value of the word at >7E00)
ABS @NT	(Computes the absolute value of the word stored at the address with label NT)

Scrolling the Screen

Many times you may find it necessary to scroll the screen. Games and application programs often use a scrolling screen to allow the user to see several pages of images or text.

In the example programs you've seen so far, whenever text has been displayed, it's been much like the DISPLAY AT statement in Extended BASIC. The screen doesn't scroll. Screen scrolling—in any of the four directions—can be handy, and even necessary. How can you program this in assembly language?

To scroll the screen up, as happens when you use the PRINT statement in BASIC, you have to read each of the 24 lines on the screen and print it one line further up.

It works like this. Once the text is displayed, the computer reads the top line and stores it in a reserved memory area. Then the second line is read and printed where the first line was. This continues until the last screen line is reached. When this happens, the first line (remember, it was stored in a reserved memory area) is printed as the bottom line and the sequence starts again. The VMBR (VDP Multiple Byte Read) utility reads the lines and the VMBW (VDP Multiple Byte Write) utility prints them again.

This next program prints HELLO on the bottom line of the screen and scrolls it up to the top. When the first screen line is passed, the scrolling sequence repeats itself. The program is listed here in several parts for clarity.

Scroll Up

First of all, the message needs to be displayed:

```
7D00        LWPI >70B8       (Load memory area for registers)
7D04        LI R0,748        (Position onscreen to display the
                             text)
7D08        LI R1,TX         (Load position in CPU RAM of
                             the text to be displayed)
7D0C        LI R2,5          (Text is five characters long)
7D10        BLWP @>6028      (Display the text)
```

Then the program reads the first line and stores it at the memory area labeled B1:

```
7D14   ST   CLR R0           (Starting position of line to be
                             read—top line, starting at position
                             0)
7D16        LI R1,B1         (Place in memory [CPU] where
                             the line read from VDP RAM will
                             be stored)
7D1A        LI R2,32         (Line is 32 characters long)
7D1E        BLWP @>6030      (Read the line into reserved mem-
                             ory area)
```

The next loop reads each of the 23 lines in turn, printing each one line up:

```
7D22        LI R0,32         (Position where to read the first
                             line on the screen to be moved—
                             it's line 2)
7D26        LI R1,B2         (Store it at reserved memory area
                             labeled B2)
7D2A        LI R2,32         (Line is 32 characters long)
7D2E   LP   BLWP @>6030      (Read the line)
7D32        AI R0,−32        (Move printing position one line
                             up to print the line which has just
                             been read)
7D36        BLWP @>6028      (Print the line)
7D3A        AI R0,64         (Move reading position two lines
                             down, to read the next line which
                             has to be moved up)
7D3E        CI R0,768        (Past the last line?)
7D42        JLT LP           (No. Stay in the scrolling loop)
```

7D44		LI R0,736	(Prepare to print top line at the bottom)
7D48		LI R1,B1	(Load position in CPU RAM where this line was stored)
7D4C		BLWP @>6028	(Print the line on the screen)
7D50		JMP ST	(Jump back to restart scrolling sequence)
7D52	TX	TEXT 'HELLO'	(Text to be displayed and scrolled)
7D58	B1	BSS 32	(Buffer where to store the top line when read)
7D78	B2	BSS 32	(Buffer where to store each line when it's moved up)
7D98		END	

When you end the program and run it, the HELLO message prints and scrolls up the screen. If you run your program from EASY BUG, the E7D00 message will scroll also, along with whatever else happens to be on the screen.

To scroll the screen down, you just have to invert the operations. The following program prints HELLO and scrolls it down off the bottom of the screen. It reappears at the top and starts over. To scroll the screen, the program reads and stores the bottom line in a designated memory area, reads line 23 and prints it as line 24, reads line 22 and prints it as line 23, and so on until the first line has been printed as the second line. Then the bottom line (read at the beginning) is reprinted as the first line.

Scroll Down

First of all, the text has to be displayed:

7D00	LWPI >70B8	(Memory area for registers)
7D04	LI R0,748	(Position on the screen where to display the text)
7D08	LI R1,TX	(Position in CPU RAM where the text is stored)
7D0C	LI R2,5	(Length of the text to be displayed)
7D10	BLWP @>6028	(Display the text)

Then the program reads the bottom line into a reserved area in CPU memory:

```
7D14  ST  LI R0,736        (Screen position to start reading
                           the bottom line)
7D18      LI R1,B1         (Position in CPU RAM to store
                           the read line)
7D1C      LI R2,32         (Length of the line to be read)
7D20      BLWP @>6030      (Read the last line from the
                           screen)
7D24      LI R0,704        (Prepare to read line 23)
7D28      LI R1,B2         (Prepare the memory area to store
                           the line)
7D2C  TB  BLWP @>6030      (Read the line from memory)
7D30      AI R0,32         (Move one line down to print in
                           its new position)
7D34      BLWP @>6028      (Print the line in its new position)
7D38      CI R0,32         (Printed line one in the position of
                           line two? Loop finished?)
7D3C      JEQ NQ           (If so, jump to label NQ)
7D3E      AI R0,-64        (No, move up two lines to read
                           the next line which has to be
                           moved one position down)
7D42      JMP TB           (Jump back to the reading and
                           printing loop)
7D44  NQ  CLR R0           (Prepare to write the last line,
                           read before, in the first line po-
                           sition of the screen)
7D46      LI R1,B1         (Load the line to be displayed,
                           stored in B1)
7D4A      BLWP @>6028      (Print it on the screen)
7D4E      JMP ST           (Start the scrolling routine again)
7D50  TX  TEXT 'HELLO'     (Text to be displayed and scrolled)
7D56  B1  BSS 32           (Memory area reserved for the
                           first line read)
7D76  B2  BSS 32           (Memory area reserved to store
                           each line as it is moved one po-
                           sition up)
7D96      END
```

The last section of the program reads each line, except the last one, and prints it one line further down. Finally, it prints line 24 in the position of the first line and restarts the scrolling.

You can also scroll the screen horizontally by moving the columns from side to side one at a time. To scroll the screen, the last column (column 31) is moved into a reserved memory area and stored. Then column 30 is moved to column 31's position, column 29 to column 30's position, and so on, until the first column (column 0) has been moved to column 1's position. Finally, column 31 is placed in column 0's position. The VSBR utility, with a loop, reads each character one at a time, so execution is considerably slower when compared to vertical scrolling. Reprinting the columns is done with the VSBW utility and a loop.

The following program fills the screen with text in BASIC and then branches to the assembly language routine to scroll the entire screen to the right. Whatever goes off the right edge reappears on the left.

Scroll Right

First of all, column 31 is read and stored:

7D00		LWPI >70B8	(Load memory area for registers)
7D04	SS	LI R6,31	(Initial screen reading position)
7D08		MOV R6,R0	(Move it to R0 for the VSBR utility)
7D0A		LI R7,B1	(Load memory area to store the bytes read from VDP RAM)
7D0E	LP	BLWP @>602C	(Start loop. Read first byte from column 31)
7D12		MOVB R1,*R7+	(Move it to the reserved memory area and increase the storage position by one byte for the next character to be read)
7D14		AI R0,32	(Move down one position to read the next character)
7D18		CI R0,768	(Still within screen limits?)
7D1C		JLT LP	(If so, return to reading loop)

With this loop, the last column has been read and stored in the CPU memory area labeled B1. Now another loop reads each column on the screen, except the last one, and prints it one column further to the right until only the first screen column remains to be updated.

7D1E		LI R6,30	(Prepare to read column 30)
7D22		LI R8,B2	(Prepare CPU memory area to store the read columns)
7D26		MOV R6,R0	(Move screen position to start reading the bytes to R0 for the VSBR utility)
7D28	L2	BLWP @>602C	(Read the byte from the screen)
7D2C		MOVB R1,*R8+	(Store it in the memory area in R8 and increase the storage position by one byte for the next character to be stored)
7D2E		AI R0,32	(Move reading position one line down)
7D32		CI R0,768	(Still within screen limits?)
7D36		JLT L2	(If so, stay in reading loop and return to L2 to read the next character)
7D38		INC R6	(Column read. Move one position to the right to print that same column just read)
7D3A		MOV R6,R0	(Move initial printing position to R0 for the VSBW utility)
7D3C		LI R8,B2	(Load the storage position of the column to be printed)
7D40	L3	MOVB *R8+,R1	(Start printing loop. Move byte to be printed into R1 for the VSBW utility and increase the pointer to the next byte to be read in R8)
7D42		BLWP @>6024	(Print the byte)
7D46		AI R0,32	(Move printing position one line down)
7D4A		CI R0,768	(Within screen limits?)
7D4E		JLT L3	(If so, column still not printed completely. Return to printing loop)
7D50		CI R6,1	(Has the second column [column 1] been printed?)
7D54		JEQ ED	(It has. Jump to end routine to print column 31 in column 0's position)

```
7D56        DECT R6           (It has not. Decrease by two the
                              value in R6 to read the new
                              column)
7D58        LI R8,B2          (Load the buffer area to store the
                              new read column)
7D5C        MOV R6,R0         (Move column read position into
                              R0 for the VSBR utility)
7D5E        JMP L2            (Jump back to read a new column)
```

Now that all columns have been moved one position to the right, all you're missing is to print column 31 in column 0's position. That's done with the following lines:

```
7D60   ED   LI R7,B1          (Load the position in CPU mem-
                              ory where the column was stored)
7D64        CLR R0            (Position to start printing the
                              column)
7D66   NQ   MOVB *R7+,R1      (Move the byte to be printed from
                              the area where the column was
                              stored to R1 for the VSBW utility.
                              Increase the pointer to the next
                              byte to be printed from R7)
7D68        BLWP @>6024       (Print the character on the screen)
7D6C        AI R0,32          (Move printing position one line
                              down)
7D70        CI R0,768         (Within screen limits?)
7D74        JLT NQ            (If so, stay in the printing loop)
7D76        JMP SS            (All columns moved. Restart
                              scrolling sequence)
7D78   B1   BSS 24            (Memory area reserved for the ini-
                              tial column read)
7D90   B2   BSS 24            (Memory area reserved to store
                              each column on the screen as it's
                              moved one position right)
7DA8        AORG >701E
701E        DATA >7FE0
7020        AORG >7FE0
7FE0        TEXT 'SCROLL'     (Add program name and position
                              to the REF/DEF Table)
7FE6        DATA >7D00
7FE8        END
```

End the program, FCTN = (QUIT), and select TI BASIC. Enter the following lines and run the BASIC program. Everything on the screen will be scrolled to the right.

```
100 CALL CLEAR
110 FOR A = 1 TO 20
120 PRINT "HOME COMPUTER"
130 NEXT A
140 CALL LINK ("SCROLL")
```

Chapter 7

BASIC and Assembly Language: A Powerful Team

BASIC and Assembly Language: A Powerful Team

Being able to link BASIC and assembly language programs greatly increases your programming power, giving you the chance to use the simplicity of BASIC with the speed and versatility of assembly language. Linking programs also gives you several ways to save the memory available for assembly language programs. You can also pass string and numeric variables between the programs.

Placing Values Directly in Memory

Before discussing the CALL LINK BASIC subroutine, we'll first see how to write and read values in memory with different BASIC statements.

Two such statements let you place integer values in CPU or VDP memory: CALL LOAD (load values into CPU RAM) and CALL POKEV (load values into VDP RAM).

CALL LOAD is used when one or more values have to be placed in CPU memory. The statement must specify a memory location (0–65535) and the value to be placed in that byte (0–255). For instance, to load the value of 3 at location 8350, you would enter:

```
CALL LOAD(8350,3)
```

If the address where you have to load the value is greater than 32767 (>7FFF), you must subtract 65536 from it and load the value at this negative address. For example, to load a value of 12 at address >9000:

>9000 = 36864 (decimal), so 36864−65536 = −28672

```
CALL LOAD(-28672,12)
```

Several values can be loaded simultaneously, starting at the address specified in the CALL LOAD. The following loads the values of 15, 139, and 252 in addresses 32000, 32001, and 32002 respectively:

```
CALL LOAD(32000,15,139,252)
```

You can also load values starting at one address and values starting at another address in one CALL LOAD statement by separating the different address and value groups with an empty string (""). For example, to load address 28532 with the value of 73 and addresses 16538, 16539, and 16540 with values 15, 19, and 251, you would use:

CALL LOAD(28532,73,"",16538,15,19,251)

You can also use CALL LOAD to pass values to an assembly language subroutine. These values must be between 0 and 255. For greater values, the best and fastest technique will be to pass the values using the CALL LINK subroutine. If you wanted to load a value of 227 (>E3) into register 7, you could first place it in location >7E00 (or any unused memory location) from BASIC, and then in assembly language, move the value at location >7E00 (32256 in decimal) into that same register. In BASIC, you'd enter:

CALL LOAD(32256,0,227)

Note that the above loads the left byte of the word at >7E00 with 0 and the right byte with 227. Thus the value in >7E00 (and >7E01) would be >00E3. In assembly language you can then move the value in >7E00 to R7 with:

MOV @>7E00,R7

You can also directly place the value in a register. As mentioned previously, all the registers store their values in a certain area of CPU RAM. If you indicated that this area should start at location >70B8, R0 would store its values in locations >70B8 and >70B9, R1 its values in >70BA and >70BB, and so on.

To load a value, in R1 for instance, you could load it directly into the corresponding memory addresses. If R1 must be loaded with >0003, you'd load >70BA (28858) with 0 and >70BB (28859) with 3:

CALL LOAD(28858,0,3)

In this way you can load values up to >FFFF. For example, to load the value >0745 into R0, you would load >07 (7) into >70B8 (28856) and >45 (69) into >70B9 (28857) with:

CALL LOAD(28856,7,69)

CALL POKEV. Values can be loaded to VDP memory in the same way by using the CALL POKEV statement. The address must be between 0 and 16383 and the value between 0 and 255. The same conditions which apply for CALL LOAD apply to the CALL POKEV instruction. With CALL POKEV, you can change values in the color tables in VDP RAM, character definitions, and in the sprite table. Values can be read or written to the screen, but the value of 96 must be added to the ASCII code of each character written to, or read from, the screen. That's due to an existing screen bias, discussed in another section.

For example, the following statement prints HELLO on the screen, starting at position 300:

CALL POKEV(300,168,165,172,172,175)

Letter	**ASCII Code**	**Code + 96**
H	72	168
E	69	165
L	76	172
L	76	172
O	79	175

Note: The CALL LOAD subroutine is also used to load an object file (assembled program) into CPU RAM. If the memory expansion is connected, the first program is loaded at starting address >A000. If the expansion is not attached, the program is loaded into the *Mini Memory* module at starting address >7118. Other programs are loaded sequentially after the first. To load the assembled program named HELLO into CPU memory from disk drive 1, for instance, you would use:

CALL LOAD("DSK1.HELLO")

Reading Values from Memory

Just as you can write integer values to specified memory addresses in CPU and VDP RAM, so you can read these values from BASIC.

To read values from CPU memory, you must use BASIC's CALL PEEK subroutine. As with previous subroutines, you can read the value from one address.

CALL PEEK(30567,X)

The above assigns the value stored at address 30567 to the BASIC variable X. Or you can read values from several addresses starting with a specified location.

CALL PEEK(28768,X,Y,Z)

assigns the values stored at locations 28768, 28769, and 28770 to the variables X, Y, and Z respectively. You can also read groups of DATA values starting at different addresses. For instance,

```
CALL PEEK(300,A,B,"",28769,RT)
```

assigns the value stored at location 300 to variable A, the value at 301 to B and the value at 28769 to RT. The "" separates nonconsecutive addresses. (Remember that to read values from addresses greater than 32767, you must first subtract 65536 to arrive at the correct *negative* address.)

To read values from VDP memory, you must use the PEEKV subroutine. The memory address from which a value is read must be between 0 and 16383. The values have a value of 96 added due to screen bias. See details of the CALL POKEV statement discussed earlier.

Initializing Memory

You can clear the 4K RAM of memory in the *Mini Memory* cartridge in much the same way as the *RE-INITIALIZE* option of the *Mini Memory* menu. This technique is useful because it allows you to clear, from BASIC, the memory in the module and in the memory expansion if it's attached. However, when this instruction is used, all DATA in memory is lost.

The memory is initialized by simply typing CALL INIT in command mode or in a program.

Using CALL LINK

Because you can link programs, you can write parts of your program—those which require speed, for instance—in assembly language. Other sections of your program you can write in BASIC. Putting the two together isn't hard.

You can write as much assembly language as you wish—you're really only limited by the amount of RAM in the *Mini Memory*. Character definitions, color assignments, title screens, and game options use up a lot of memory, so it's probably best to write these in BASIC. These operations generally don't affect the program's speed anyway.

To link a BASIC and assembly language program, you'll use the BASIC subroutine CALL LINK. The assembly language program you're calling must be loaded in *Mini Memory* RAM or in the memory expansion unit and must have its name and position added to the REF/DEF Table (as you saw in Chapter 4). You can call several different assembly language routines, as long as each one has its own entry in the REF/DEF Table.

When your BASIC program is ready to link with the assembly language program, enter:

```
CALL LINK("program name")
```

where *program name* is one to six characters long and is the name of the program as listed in the REF/DEF Table. Execution continues automatically when the assembly language program is called. If the program is not found in the table, the error message PROGRAM NOT FOUND will appear.

When control is passed to the assembly language program, the memory area for the registers is automatically loaded, starting at location >70B8. You don't need to load this area with the LWPI instruction. (This is also true for the *Mini Memory* RUN option, but not for EASY BUG's EXECUTE command.)

The BASIC program address where execution will return once the assembly language routine is finished (assuming you *want* the program to return to BASIC) is stored in register 11, so care must be taken not to lose this value. If you need to use R11 for your own program (if you're using subroutines in your assembly language program, R11 is used to store the return address of the subroutines), you can move the value in R11 to another register or memory location and store it until you're ready to return to BASIC. It could be done like this:

```
MOV R11,R14
```

or

```
MOV R11,@>7F00
```

When you're ready to return to BASIC, you must branch to the value stored in R11 (or whatever other register you put the return address in). If you stored the return address at a memory address, move it back to any register. To branch to the value stored in a register, precede it by an asterisk. For example, to branch back to BASIC, assuming the return address is loaded in R11, you would enter:

```
B *R11
```

Or if the return address was stored in location >7F00:

```
MOV @>7F00,R5
B *R5
```

When this instruction is encountered, control is immediately transferred back to the BASIC program, specifically to the statement following the subroutine call.

Linking Dangers

You'll have to keep certain things in mind when you're linking programs. First of all, before returning from the assembly language program to BASIC, you must clear the status byte (byte >837C). If this isn't done, a false error might be reported upon return to BASIC. Clear this byte with:

```
CLR @>837C
```

Another problem is an existing screen bias of >60 (96 in decimal). This means that whatever character is read from the screen or displayed on the screen must have the value of >60 added to its hexadecimal ASCII code. This screen bias is also present when using the POKEV and PEEKV subprograms to read and display values on the screen from BASIC. You must always add 96 decimal to the ASCII code of the character you are printing or reading from the screen.

For instance, suppose you want to display the at (@) symbol on the screen in position 300. The assembly language program will be called from BASIC. First you would have to add the screen bias value of >60 to the code of the @ symbol (>40) and then display it on the screen. The code would then be >A0 (>40 + >60) and the program segment to display the symbol would be:

```
LI R0,300
LI R1,>A000
BLWP @>6024
```

The same thing happens when you read a value from the screen in an assembly language program called from BASIC. If you read a code from the screen and want to check if the character was the exclamation mark (!), with ASCII code >21, you would compare the value read from the screen to >81 (>21 + >60).

```
LI R0,300
CLR R1
BLWP @>602C
CI R1,>8100
```

If you're writing an assembly language program to be called from BASIC and you want to display text on the screen, do not use the TEXT directive to place the string in memory. First make a list of the ASCII codes of each character in the string (spaces included) and add 96 (>60) to each one. Then add the string to memory with the DATA directive.

Letter	**ASCII Code**	**Code + 96**
H	72	168
E	69	165
L	76	172
L	76	172
O	79	175

To place the text in memory:

DATA 168, 165, 172, 172, 175

Remember that now the text will be displayed correctly on the screen only if the assembly language program is called from BASIC.

To finish, if you have enabled VDP interrupts with the LIMI 2 instruction to move sprites, generate sounds, and so on, disable them with the LIMI 0 instruction before you return to BASIC, unless you need them enabled for a specific reason (like simultaneous sound and BASIC program execution).

Linking Two Programs

The following program defines character 128 in BASIC, sets its color, and then links to assembly language. The assembly language program displays the character on the screen and then returns to BASIC where the message READY—RETURNED FROM ASSEMBLY LANGUAGE displays and program execution stops with an endless loop. The only purpose of this program is to clarify the linking technique. First enter the assembly language program, then the BASIC program.

Linking—Assembly Language Listing

```
7D00      LI R0,300         (Load character printing position)
7D04      LI R1,>E000       (Load ASCII code of character to
                            be printed [plus >60 to solve the
                            screen bias], since the program
                            will be called from BASIC)
7D08      BLWP @>6024       (Print the byte on the screen)
```

```
7D0C      CLR @>837C        (Clear the status byte to avoid
                            false errors reported upon return
                            to BASIC)
7D10      B *R11            (Return to BASIC)
7D12      AORG >701E
701E      DATA >7FE0
7020      AORG >7FE0        (Add name and starting address of
                            the assembly language program to
                            the REF/DEF Table)
7FE0      TEXT 'LINKS '
7EE6      DATA >7D00
7FE8      END
```

Linking—BASIC Listing

```
100 CALL CLEAR
110 CALL CHAR (128,"FF818181818181FF")
120 CALL COLOR (13,5,1)
130 CALL LINK ("LINKS")
140 PRINT "READY—RETURNED FROM ASSEMBLY
LANGUAGE"
150 GOTO 150
```

Now run the BASIC program. A blue square will appear in screen position 300, indicating that control was successfully passed to the assembly language program. The message READY—RETURNED FROM ASSEMBLY LANGUAGE displays when control is returned to the BASIC program.

You can see that all character definitions you'll use in your program can be written in BASIC, thus avoiding using your valuable assembly language CPU RAM to program the definitions.

Passing Data Between Programs

The LINK subroutine not only lets you combine the advantage of both BASIC and assembly language, it also allows you to pass data between programs. You may pass the values of numeric or string variables, direct numbers, elements from an array, or an entire array if necessary. Up to 15 variables may be passed between programs with the LINK subroutine. In the BASIC program you only have to specify the variables you want to use for the data transfer in the CALL LINK statement. In the assembly language program, you have to use one of five TI BASIC interface utilities to retrieve a number or string in the assembly language program, pass a number or string to BASIC from assembly language, or report an error. The utilities are: NUMREF, NUMASG, STTREF, STRASG, and ERR. They'll be discussed after the following section.

Changing Floating-Point and Integer Values

Any number which is transferred between an assembly language and a BASIC program must be in floating-point format. Since most values in your program will be in integer form, you can use two ROM routines to change integer values to floating-point values and vice versa.

To change a floating-point value to an integer value, the value must be loaded starting at address >834A. Then you simply branch to the ROM routine in >601C, indicating the routine to be executed with a DATA statement following the branch. To convert a floating-point value to an integer value, use DATA >1200. The integer result (>FFFF maximum) will be placed at location >834A. For example, assuming the floating-point value you want to change starts at >834A, the following program segment would leave its equivalent integer value loaded in register 7:

```
BLWP @>601C
DATA >1200
MOV @>834A,R7
```

To change an integer value to a floating-point value, you also use a ROM routine (not the one stated in the *Mini Memory* manual, with DATA >2300, as this routine only works with the *Editor/Assembler* cartridge). A routine which executes the desired conversion is located in the ROM of the *Mini Memory* cartridge and is accessed by branching to a ROM routine with DATA >7200. In other words, if your integer value is located at >834A, all you have to do is branch to execute the ROM routine

```
BLWP @>601C
```

and follow the branch with the value of >7200.

```
DATA >7200
```

The floating-point equivalent of the integer value will then be placed starting at >834A. For instance, if the integer value is located at >834A, the program segment:

```
BLWP @>601C
DATA >7200
```

leaves the floating-point equivalent starting at >834A. In the examples which follow, these routines will be used when values are transferred between BASIC and assembly language programs. Remember also that for all mathematical routines in ROM and GROM which require floating-point values, you'll have to convert your integer values to floating-point values first.

Numeric Values from BASIC to Assembly Language

To pass a numeric value from a BASIC program to an assembly language program, you must use the NUMREF (get numeric parameter) utility stored in >6044. In the BASIC program, the variables you want to use to pass data are listed after the program name.

CALL LINK (*"program name"*,S,TR,NS,H$)

S is considered the first variable, TR the second, NS the third, and H$ the fourth. If you need to know the value of a variable, say NS, use the NUMREF utility in assembly language because it's a numeric variable.

The assembly language program is prepared to receive the value of NS like this: The NUMREF utility reads the value of a specified variable and places the value in floating-point format starting at location >834A. The routine requires R0 loaded with the array element number if the variable belongs to an array (otherwise just place a zero), and R1 loaded with the variable's position in the list of variables in the LINK subprogram. To read the value of NS, for instance, you'd load a 3 in R1, since NS is the third variable. Then, by branching to >6044, the value of the variable is placed starting at address >834A. If you need the value in integer format, use the ROM routine to convert the floating-point value to an integer value.

The following program will call an assembly language program from BASIC and pass the value of a numeric variable to it. In assembly language, this value will be read and then converted from a floating-point value to an integer value. The integer value will then be assigned to R5, as well as placed at >7F00, and control will then return to BASIC.

Numeric Values—Assembly Language

```
7D00    CLR R0              (Variable does not belong to an
                            array)
7D04    LI R1,1             (Use first variable of the vari-
                            able list in the LINK statement)
7D08    BLWP @>6044         (Get numeric parameter
                            routine)
7D0C    BLWP @>601C         (Convert floating-point to
                            integer)
7D10    DATA >1200          (ROM routine to be executed)
7D12    MOV @834A,@>7F00    (Move the integer result in
                            >834A to memory address
                            >7F00)
7D18    MOV @>834A,R5       (Place the integer value in R5)
7D1C    CLR @>837C          (Clear the status byte to avoid
                            false errors upon return to
                            BASIC)
7D20    B *R11              (Return to BASIC)
7D22    AORG >701E
701E    DATA >7FE0
7020    AORG >7FE0          (Add name and position of the
                            program to the REF/DEF
                            Table)
7FE0    TEXT 'NUMVAL'
7FE6    DATA >7D00
7FE8    END
```

Numeric Values—BASIC Program

```
100 X=18
110 CALL LINK("NUMVAL",X)
120 END
```

When you run the BASIC program, the value of 18 will be assigned to variable X and control transferred to the assembly language program. The assembly language program will read the value of X and place its value in floating-point format starting at >834A. The value could occupy up to eight bytes, starting at that address.

This value is then converted to an integer value and placed in the memory word at >834A. It's moved to register 5 *and* to address >7F00. The status byte is cleared to avoid false errors and control returned to BASIC, where the program ends.

To know whether the value of X was read correctly in the assembly language program, you can check the value stored in >7F00. It should be >0012. To do this, FCTN = (QUIT) and select EASY BUG. Skip the title screen by pressing any key and use the M command to display CPU memory by typing *M7F00* and pressing ENTER. CPU memory addresses starting at address >7F00 will be displayed. The first value you will see is >00, the left byte of the word stored at >7F00. Press ENTER again and you'll see the value of >12. In other words, >7F00 has been loaded with >0012 (18), which is the value you passed from BASIC.

Passing Values Back to BASIC

The variable list in the BASIC CALL LINK statement can also be used to retrieve values from the assembly language program. Any variable in the list can be used. The assembly language program assigns the values to determined variables from the list in the CALL LINK statement. The same variable can be used to pass a value to assembly language and *then* to retrieve a new value.

In the assembly language program, the variable chosen is assigned its value. This is done with the NUMASG (NUMeric ASsiGnment) utility at location >6040. The routine requires R0 loaded with the array element number if the variable belongs to an array (if not, just clear the register), and R1 loaded with the variable position the value will be assigned. The value assigned to the variable must be loaded starting at location >834A and in floating-point format. If you've been working with integer values, convert the integer number into floating-point format first. Once this is done, and R0 and R1 have been loaded with the appropriate values, you can branch to execute the NUMASG routine in >6040. When you return to BASIC, the variable will have the value assigned by the assembly language program.

The following example program links to an assembly language program, where the value 15736 will be assigned to the variable X1. Control will then pass to the BASIC program which will print the value on the screen. Enter the assembly language listing first.

Passing Back to BASIC—Assembly Language

```
7D00    LI R7,15736       (Load value to be passed to
                          BASIC)
7D04    MOV R7,@>834A     (Place it at >834A)
7D08    BLWP @>601C       (Branch to ROM routine)
7D0C    DATA >7200        (Convert the integer value to
                          floating-point format for the value
                          transfer to BASIC. The converted
                          value will be stored starting at
                          >834A, ready for the NUMASG
                          routine)
7D0E    CLR R0            (Variable does not belong to an
                          array)
7D10    LI R1,1           (Assign the value to the first vari-
                          able of the variable list in the
                          BASIC CALL LINK statement)
7D14    BLWP @>6040       (Execute the numeric assignment
                          routine, thus assigning the value
                          to the BASIC variable)
7D18    CLR @>837C        (Clear the status byte so no false
                          errors will be reported upon re-
                          turn to BASIC)
7D1C    B *R11            (Return to BASIC)
7D1E    AORG >701E        (Add the name and position of
                          the program to the REF/DEF
                          Table)
701E    DATA >7FE0
7020    AORG >7FE0
7FE0    TEXT 'TEST
7FE6    DATA >7D00
7FE8    END
```

Passing Back to BASIC—BASIC Program

```
100 CALL CLEAR
110 CALL LINK ("TEST",X1)
120 PRINT X1
130 END
```

When you run the BASIC program, the screen will be cleared and control transferred to the assembly language program. The latter will assign the value of 15736 to the variable X1 in the CALL LINK statement. Control will then return to BASIC, where the value of the variable will be printed so you can see that it was transferred correctly.

Operating with Strings

Strings can be passed between programs quite easily, but you must remember the existing screen bias of >60 (96 in decimal) when passing a string to assembly language. When a string is passed from assembly language to BASIC, there's no screen bias to worry about.

First of all, you'll see how to pass a string from BASIC to assembly language and how to handle problems arising from the existing screen bias. Then we'll look at passing strings from assembly language to BASIC.

BASIC to assembly language. The BASIC statement used to pass a string is the same as for that used to pass numeric variables. Just include the string variable(s) in the variable list found in the CALL LINK statement.

CALL LINK("*program name*",G,H$)

where H$ is the string variable.

To retrieve the string in the assembly language program, you must use the STRREF (get string parameter) utility in location >604C to place the string in an assigned memory area. Before executing this routine, you first need to prepare a memory area to store the string which will be read. You can do this with the BSS directive. STRREF also requires R0 to be loaded with the array element number if the string belongs to an array, or zero if it does not. Load R1 with the position of the variable you want to use in the CALL LINK statement, just as in previous examples. Register 2 must be loaded with the starting address in CPU memory where you want the string to be stored. The first byte of this memory area must contain the string's maximum length. If the string doesn't fit into the assigned area (determined by this first byte of the reserved area, also called the *length byte*), an error is reported. Assuming the string *does* fit, it's located and the length byte is updated to reflect the correct length of the string. This is useful because you can then know the exact length of the string you have in memory—something needed to later display the text or string.

The following program passes a string from BASIC to assembly language. Note that once the string has been stored in memory, it cannot be displayed using the VMBW (VDP

Multiple Byte Write) routine because the screen bias value of >60 has to be added to the ASCII code of each character before it's displayed. In the example program, a loop reads the characters from the string one by one, adds >60 to the ASCII code of each character, and then displays it on the screen using the VSBW (VDP Single Byte Write) utility.

String Passing—Assembly Language

7D00	CLR R0	(Clear R0. The string does not belong to an array)
7D02	LI R1,1	(Use the first variable of the variable list in the CALL LINK statement)
7D06	LI R2,RM	(String has to be placed at the reserved CPU memory area labeled RM which will be added to the end of the program)
7D0A	LI R5,>1200	(Prepare to write the maximum string length to the first byte of the reserved memory area. The maximum string length will be 18 [>12] characters)
7D0E	MOVB R5,@RM	(Move the maximum length byte to the first byte of the reserved memory area)
7D12	BLWP @>604C	(Place the string in the prepared memory area with string parameter routine. The length byte is updated if the string fits in the memory area. Otherwise, an error message is reported)
7D16	LI R0,296	(Load screen position where the text must be displayed)
7D1A	MOVB *R2+,R5	(Move the new length byte to R5 to keep track of how many characters have been displayed. Add one to the pointer to the screen in memory, so the next character loaded will be the first one from the string)
7D1C	SWPB R5	(Place the length of the string in the right byte of R5)

```
7D1E  LT  MOVB *R2+,R1     (Start loop to display the string.
                           Move the ASCII code of the first
                           character to be printed into the
                           left byte of R1, increasing R2 to
                           point to the next character in the
                           string at the same time)
7D20      AI R1,>6000      (Add the screen bias value of >60
                           to the ASCII code of the
                           character)
7D24      BLWP @>6024      (Print the character on the screen)
7D28      INC R0           (Increase the screen printing
                           position)
7D2A      DEC R5           (Decrease number of characters
                           left to display)
7D2C      JNE LT           (If there still are characters left to
                           display, stay in the loop)
7D2E      CLR @>837C       (Clear the status byte, preparing
                           to return to BASIC)
7D32      B *R11           (Return to BASIC)
7D34  RM  BSS 20           (Reserve memory for the string)
7D48      AORG >701E
701E      DATA >7FE0
7020      AORG >7FE0       (Add the name and position of
                           the program to the REF/DEF
                           Table)
7FE0      TEXT 'STRING'
7FE6      DATA >7D00
7FE8      END
```

Using the VSBW utility to display the string, you can correct the ASCII code of each character and also control the screen position, which is updated after each character is displayed.

String Passing—BASIC Program

```
100 CALL CLEAR
110 R$="ASSEMBLY LANGUAGE"
120 CALL LINK("STRING",R$)
130 GOTO 130
```

When you run the BASIC program, the string will immediately be displayed. If the string you transfer to the assembly language program is shorter, it will still be displayed correctly on the screen. If the string is longer than 18 characters, the value of the length byte, the message STRING TRUNCATED IN 120 will appear and program execution will be interrupted.

Passing a String Back to BASIC

This problem is much simpler than the previous one because you don't need to worry about screen bias. To pass a string *back* to BASIC, use the STRASG (STRing ASsiGnment) utility at location >6048. The routine requires that the string is loaded in a predetermined CPU memory area and that the first byte of that area contains the length of the string. R0 must then be loaded with the array element number if the string is to be assigned to an array element, or zero if not. R1 must contain the list number of the variable you want to assign the string to, and R2 must contain the starting address of the string in CPU RAM. When you branch to the STRASG routine, the string will be assigned to the desired variable.

The next program links a BASIC program to an assembly language program, where a string will be assigned to a variable. The string passed from assembly language will then be printed on the screen in BASIC.

Back to BASIC—Assembly Language

```
7D00      LI R5,>0C00      (Load the length of the string in
                           the left byte of R5. The string is
                           12 characters long)
7D04      MOVB R5,@RM      (Write the length byte to the first
                           byte of the reserved memory area)
7D08      CLR R0           (String is not an array element)
7D0A      LI R1,1          (Assign the string to the first vari-
                           able in the variable list)
7D0E      LI R2,RM         (The string is located starting at
                           RM)
7D12      BLWP @>6048      (Execute the string assignment
                           routine)
7D16      CLR @>837C       (Clear the status byte to avoid
                           false errors upon return to BASIC)
7D1A      B *R11           (Return to BASIC)
7D1C  RM  TEXT ' HELLO THERE ' (String to be displayed.
                           The first byte is left blank because
                           it will be replaced by the length
                           byte)
```

```
7D2A      AORG >701E
701E      DATA >7FE0
7020      AORG >7FE0
7FE0      TEXT 'TRYOUT'   (Add name and position of the
                          assembly language program to the
                          REF/DEF Table)
7FE6      DATA >7D00
7FE8      END
```

Back to BASIC—BASIC Program

```
100 CALL CLEAR
110 CALL LINK("TRYOUT",VR$)
120 PRINT VR$
130 GOTO 130
```

When you run the BASIC program, the message HELLO THERE will be printed on the screen.

Generating Error Messages

You can report a TI BASIC error upon return from the assembly language program by using the ERR (ERRor reporting) utility stored in >6050. This utility is used by simply loading the desired error code in the left byte of R0 and branching to execute the routine. The list of error codes you can issue is found on page 55 of the *Mini Memory* manual.

For example, to report a NUMBER TOO BIG error (code >14) upon return to BASIC from assembly language, you would use something like the following program.

Errors

```
7D00      LI R0,>1400     (Load code of the error to be re-
                          ported in the left byte of R0)
7D04      BLWP @>6050     (Execute the error-reporting rou-
                          tine in >6050)
7D08      CLR @>837C      (Clear the status byte)
7D0C      B *R11          (Return to BASIC, where the error
                          will be reported)
7D0E      AORG >701E
701E      DATA >7FE0
7020      AORG >7FE0      (Add name and position of the
                          program to the REF/DEF Table)
```

```
7FE0        TEXT 'ERROR '
7FE6        DATA >7D00
7FE8        END
```

For the error to be reported, simply link to the assembly language program. When the program control is passed back to BASIC, the error message is reported. You can type in direct mode:

CALL LINK ("ERROR") and press ENTER

Displaying Strings

The following assembly language program allows you to display a string at a specified screen position. No screen limit checks are executed, so if the message printed goes off the screen, it will be printed in other VDP memory areas. Also, passing a null string to the assembly language program will make the program run incorrectly and cause the computer to *lock up* (it won't respond to keypresses). You'll have to turn the computer off and on again to regain control.

Since there's a screen bias of >60, the program has to read each character, add >60 to its hexadecimal ASCII code, and then display it on the screen.

Displaying a String—An Example

7D00	CLR R0	(Clear R0. Variable does not belong to an array)
7D02	LI R1,1	(Read the value of the first variable in the variable list [the screen position to display the string])
7D06	BLWP @>6044	(Read the value of the numeric variable and store it in floating-point format starting at >834A)
7D0A	BLWP @>601C	(Execute ROM routine)
7D0E	DATA >1200	(Convert floating-point value to integer value)
7D10	MOV @>834A,R7	(Place the text screen position in register seven)
7D14	LI R1,2	(Prepare to read the string variable, second variable in the variable list)
7D18	LI R2,RS	(The string will be placed at the CPU RAM area assigned label RS)

```
7D1C       LI R5,>8000      (Prepare to write the maximum
                            string length allowed to the first
                            byte of the reserved memory area.
                            The maximum string length will
                            be >80, or 128 in decimal)
7D20       MOVB R5,@RS      (The string will be placed at the
                            CPU RAM area assigned RS)
7D24       BLWP @>604C      (Read the string from BASIC)
7D28       MOV R7,R0        (Prepare to print the string. Move
                            the string printing position to R0
                            for the VSBW utility)
7D2A       CLR R4           (Prepare R4 to keep track of the
                            characters left to be displayed in
                            the string)
7D2C       MOVB *R2+,R4     (Move the length byte to R4, and
                            increase the pointer to the string
                            to the first character of the string)
7D2E       SWPB R4          (Place the length byte in the right
                            byte of R4)
7D30  LP   MOVB *R2+,R1     (Move the ASCII code of the
                            character to be printed to the left
                            byte of R1 and increase the
                            pointer in R2 to the next character
                            of the string)
7D32       AI R1,>6000      (Add the screen bias value to the
                            ASCII code of the character)
7D36       BLWP @>6024      (Display the character)
7D3A       INC R0           (Increase printing position)
7D3C       DEC R4           (Decrease number of bytes left to
                            be displayed)
7D3E       JNE LP           (If there are still characters left,
                            stay in the printing loop)
7D40       CLR @>837C       (String displayed. Clear the status
                            byte to avoid false errors upon re-
                            turn to BASIC)
7D44       B *R11           (Return to BASIC)
7D46  RS   BSS 128          (Memory area for the string)
7DC6       AORG 701E
701E       DATA 7FE0
7020       AORG 7FE0
7FE0       TEXT 'UTIL       (Add name and position of the
                            program to the REF/DEF Table)
```

```
7FE6        DATA 7D00
7FE8        END
```

To use the assembly language text-displaying program, use the following BASIC statement:

CALL LINK("UTIL",X,M$)

where X is the screen position from where to display the string (0–767) and M$ is the string to be displayed (the maximum length of the string is 128 characters). For example:

CALL LINK("UTIL",100,"HI THERE")

displays the message HI THERE on the fourth line of the screen.

An Assembly Language Square Root

The following BASIC program asks for a decimal value and then links to the assembly language program which calculates the square root of the number. It uses the SQR floating-point GROM routine. The answer is then returned to BASIC and printed. The assembly language routine executes the same as the BASIC statement SQR and is really only here as a programming example.

Square Root—Assembly Language Routine

7D00	CLR R0	(Value does not belong to an array)
7D02	LI R1,1	(Read the value of the first variable in the CALL LINK variable list)
7D06	BLWP @>6044	(Execute the NUMREF routine)
7D0A	BLWP @>6018	(Execute GROM routine)
7D0E	DATA >0026	(Find the square root of the floating-point value stored starting at >834A)
7D10	BLWP @>6040	(Return the new value to the same variable in the CALL LINK statement)
7D14	CLR @>837C	(Clear the status byte)
7D18	B *R11	(Return to BASIC)
7D1A	AORG >701E	
701E	DATA >7FE0	

```
7020    AORG >7FE0
7FE0    TEXT 'SQROOT'    (Add name and position of the
                         program to the REF/DEF Table)

7FE6    DATA >7D00
7FE8    END
```

You can find the square root of a value with the following BASIC lines:

```
100 CALL CLEAR
110 INPUT A
120 CALL LINK("SQROOT",A)
130 PRINT A
140 GOTO 110
```

Or directly, in command mode, with:

```
A=24
CALL LINK("SQROOT",A)
PRINT A
```

which calculates the square root of 24 and prints it on the screen.

Chapter 8

Character Definitions and Color Changes

Character Definitions and Color Changes

Character definitions and colors are some of the most important parts of programming games. They can be just as important when you're writing application programs, or programming utilities, for you may want to use custom characters or snappy color changes to make your program look more professional. Using character definitions and changing colors *can* be done from BASIC, but they can be done much faster, and even easier, if you're using assembly language.

Where *Are* the Definitions?

In assembly language, you can change the definition of characters just as you can in BASIC. All redefined characters remain redefined as long as the program is working. Only when the program stops and the computer returns to command mode do some revert to their original shapes. Characters with ASCII values greater than 127 remain defined when a program stops running, just as in BASIC. In assembly language, other characters, such as 30 (cursor) and 31 (edge character), remain defined when a program stops running. This means that you can give the cursor and border character the shape you like and let them remain that way until you type NEW. You'll be seeing this in an example program.

All characters must have their definitions stored *somewhere* in memory. The first step is to know where those definitions are stored.

A character is described by 16 hexadecimal digits, as you know from BASIC. Since each byte contains two digits, you need eight bytes for a character. Remember that you use the DATA directive to place the definition into memory. For example, to define a solid box character, you would enter:

```
DATA >FFFF,>FFFF,>FFFF,>FFFF
```

This places the 16-digit definition in memory. Eight bytes of memory are used. The definition is still not assigned to any character, however. All character definitions are kept in a table, located in VDP memory. If a program runs entirely in assembly language, this table is in one place in memory. If the program is called from BASIC, the table is in another area of memory.

Let's suppose a program is written entirely in assembly language and is executed either with the *Mini Memory* RUN option or EASY BUG's EXECUTE option. If that's the case, then the table is located from >0800 to >0FFF (see Appendix E, page 75, of the *Mini Memory* manual), though the default character set goes from >0900 to >0AFF. In other words, because each character definition occupies eight bytes:

Character 32 occupies memory addresses >0900 through >0907
Character 33 occupies memory addresses >0908 through >090F
Character 34 occupies memory addresses >0910 through >0917

.
.
.

If you want to change the definition of a character in the program, character 35 for instance, you would load the definition somewhere in CPU memory, outside program execution. To assign the definition to a character, you'd use the VMBW routine to write the definition to VDP RAM (which is where the table is located).

You first need to find the place in VDP RAM where the character is to go. Character 35 occupies bytes >0918 through >091F in the table. You would then have a program segment which looks like this:

```
LI R0,>0918      (Location in VDP RAM to start writing the
                 character definition)
LI R1,DF         (Location in CPU RAM of the new definition)
LI R2,8          (Number of bytes to write to VDP RAM. The
                 definition is eight bytes long)
BLWP @>6028      (Write bytes to the Character Table in VDP
                 RAM)
  .
  .
  .
DF DATA >FF81,>8100,>0081,>81FF (Character definition)
```

When executed, this program segment assigns the definition loaded at label DF to character 35.

All characters are defined in this way. But you don't need to write 16 routines to define 16 characters, if they're consecutive characters. For example, to define characters 35, 36, and 37, just load R0 with the starting location of the definitions.

Because you're starting with character 35, you'll load R0 with >0918. In R1 you'll load the position in text where the character definitions are located. Remember these definitions must be consecutive and in order. That is, the first definition in memory will be assigned to character 35, the second to character 36, and the third to character 37.

In R2 you then load the number of bytes to write to VDP RAM. There are three characters being defined simultaneously. That means you'll need 24 (3 × 8) bytes.

```
     LI R0,>0918    (Location in VDP memory of the definition of
                    the first character [35])
     LI R1,DF       (Location in CPU RAM of the new definitions)
     LI R2,24       (Number of bytes to write to the Character
                    Table)
     BLWP @>6028    (Write definitions to the table)
       .
       .
       .
DF DATA >FF81,>8100,>0081,>81FF (Character definitions)
   DATA >0101,>0101,>0101
   DATA >0101,>0404,>7575,>C2C2,>C2C2
```

The same technique applies if an assembly language program is to be called from BASIC. The only difference is the location of the Character Table in VDP memory. The table is located from >0400 to >05FF (1024 to 1535 decimal). See Appendix F, page 76, in the *Mini Memory* manual. This means that:

Character 32 occupies bytes >0400 through >0407
Character 33 occupies bytes >0408 through >040F
Character 34 occupies bytes >0410 through >0417

.
.
.

To change the definition of a character, work out its position in the table and use the methods already described.

Changing Cursor

This example uses all you've just seen. It's an assembly language program that, when called from BASIC, changes the definition of the cursor (ASCII code 30). (Remember that the cursor does *not* return to its original definition when a program stops execution. That makes this program possible.)

Because the program is called from BASIC, the definition of character 32 is located from >0400 to >0407. Character 31 then occupies bytes >03F8 through >03FF, and character 30 (the cursor), bytes >03F0 through >03F7. Your redefined cursor has to be written into VDP RAM starting at >03F0. The following program redefines the cursor, making it look like a small, empty box.

Redefined Character

```
7D00       LI R0,>03F0        (Location in VDP memory of the
                              definition for character 30)
7D04       LI R1,CS           (Position in CPU memory where
                              the new definition is added. Label
                              CS used)
7D08       LI R2,8            (Number of bytes in the
                              definition)
7D0C       BLWP @>6028        (Write definition to the Character
                              Table)
7D10       CLR @>837C         (Clear status byte to avoid false
                              errors upon returning to BASIC)
7D14       B *R11             (Return to BASIC)
7D16   CS  DATA >FF81,>8181,>8181,>81FF (Character
                              definition)
7D1E       AORG >701E
701E       DATA >7FE0
7020       AORG >7FE0
7FE0       TEXT 'CURSOR'      (Add name and position of the
                              program in the REF/DEF Table
                              for the program to be called from
                              BASIC)
7FE6       DATA >7D00
7FE8       END
```

After adding the program's name and position to the REF/DEF Table, end it and press FCTN = (QUIT). Select TI BASIC. In command mode type:

CALL LINK("CURSOR") and press ENTER

Look at the cursor now. If you want a different shape, simply return to the *Assembler* and change the definition added with label CS to the definition you want. The cursor will return to its normal state when you turn off the computer, enter FCTN = (QUIT), or type NEW.

Because the Character Table is in VDP memory, you can also change definitions directly with BASIC's CALL POKEV statement, which lets you POKE values directly into VDP RAM. To change the cursor definition, you'd do the following procedure.

The cursor definition starts at address >03F0 in VDP RAM, which is 1008 in decimal. That's where you'll POKE the new cursor definition. If you wanted to assign the following definition to the cursor,

```
FF818181818181FF
```

you'd first divide the digits forming the definition into groups of two, in this way:

```
FF-81-81-81-81-81-81-FF
```

Then find the decimal equivalent of each hexadecimal two-digit group:

```
>FF = 255
>81 = 129
>81 = 129
>81 = 129
>81 = 129
>81 = 129
>81 = 129
>FF = 255
```

Finally, POKE the decimal values into memory:

```
CALL POKEV(1008,255,129,129,129,129,129,129,255)
```

By typing this one statement in command mode and pressing ENTER, the cursor definition will be changed.

Changing Colors

Colors are changed much like character definitions. Data for the colors of each character set (a character set is a group of eight sequential characters) is kept in the Color Table, which is also located in VDP memory. If the program is called from BASIC, this table is in one memory area. However, if it's an assembly language program, the table is found someplace else. Look in Appendix E in your *Mini Memory* manual and you'll see that the Color Table is found from location >0380 to location >03FF (896–1023 in decimal) when the program works entirely in assembly language. If the program is called from BASIC, the table starts at >0300 (768).

The color for each character set is represented by just one byte. The left digit of the byte represents the foreground color and the right digit the background color. See Appendix D for a list of hexadecimal color codes.

For example, the byte >18 represents a character set with the foreground color black (>1) and background color medium red (>8).

Changing the color of a character set in assembly language involves finding the set's byte in the Color Table in VDP RAM and writing the byte with the desired color values.

Remember that if you're working entirely in assembly language, the color table starts at >0380. Thus:

Location >0380 corresponds to characters 0 through 7
Location >0381 corresponds to characters 8 through 15
Location >0382 corresponds to characters 16 through 23

.
.
.

If you have an assembly language program called from BASIC, the Color Table starts at location >0300. This makes things a bit trickier. The bytes corresponding to the various character sets are:

Character set 1 (characters 32–39) is controlled by byte >0310
Character set 2 (characters 40–47) is controlled by byte >0311

.
.
.

To change the color of characters 56 through 63 (character set 4) in an assembly language program, you'd use a segment something like:

LI R0,>0387	(Byte corresponding to character set 4 in the color table)
LI R1,>EF00	(Color byte to be written, in the left byte of R1. The color is gray [>E] on white [>F])
BLWP @>6024	(Write byte to the table)

You can also assign colors to several character sets simultaneously by using the VMBW utility. The following program, when called from BASIC, will change the color of character sets 5–8, and make each one a different color combination. Remember that these example programs can also be written directly with BASIC subroutines like CALL POKEV. We're creating them here so that you can get an idea of how the Color Table and Character Table are changed.

Color Changing

This program is called from BASIC, so the Color Table starts at location >0300. Location >0310 controls the color of character set 1, so:

Character set 5 is controlled by location >0314
Character set 6 is controlled by location >0315
Character set 7 is controlled by location >0316
Character set 8 is controlled by location >0317

The colors to be assigned will be stored in the memory area with label CL and will be used with the VMBW utility.

Colors

```
7D00      LI R0,>0314          (Address in the table where the
                               first character set to have its
                               color changed is located)
7D04      LI R1,CL             (Address in memory where the
                               color bytes to be assigned are
                               found)
7D08      LI R2,4              (Number of character sets to
                               have their color changed)
7D0C      BLWP @>6028          (Write color bytes to the Color
                               Table in VDP)
7D10      CLR @>837C           (Clear status byte to avoid false
                               errors upon return to BASIC)
7D14      B *R11               (Return to BASIC)
7D16  CL  DATA >C8E1,>4F63     (Color bytes)
7D1A      AORG >701E
701E      DATA >7FE0
7020      AORG >7FE0           (Add name and position of pro-
                               gram to the REF/DEF Table)
7FE0      TEXT 'COLOR '
7FE6      DATA >7D00
7FE8      END
```

End the program, press FCTN = (QUIT), and select TI BASIC. Type in and run the following program:

```
100 CALL CLEAR
110 CALL LINK ("COLOR")
120 PRINT "A - I - R - Z"
130 GOTO 130
```

The assembly language program will make the characters in character set 5 dark green on medium red (>C8); those in character set 6 will be gray on black (>E1); character set 7 will appear as dark blue on white (>4F); and those in character set 8 will change to dark red on light green (>63). The BASIC program links to the assembly language program, which changes the colors of the sets and then returns to BASIC. A character from each of the sets is printed so you can see that the color has been changed. An endless loop stops the program so that the colors will not revert to their original shades.

Changing Screen Color

The screen color is changed in an entirely different way. Screen color is also represented by one byte. To turn the screen completely black (>1), for instance, you'd use byte >11 (black on black).

So far you've been writing data to VDP RAM, an area of memory which holds information for the VDP (Video Display Processor) chip. However, there are also special control locations *within* the VDP chip hardware, called the *VDP write-only registers* (see Chapter 11). These registers control a number of display functions, including screen color. The register which controls the screen color is VDP register 7.

To change the value in one of the VDP registers, you must use a system utility routine, VWTR (VDP Write to Register), which was mentioned in Chapter 2. To use this routine, R0 (the register 0 you've been using all along) must have its left byte loaded with the VDP register number you want to change (in this case, >07) and the right byte loaded with the value you want to put in that register (>11 for this example, to turn the screen black). Then you just branch to the VWTR routine at location >6034.

So, to make the screen completely black, you'd use a program segment like this:

```
LI R0,>0711        (Load left byte of R0 with the VDP register
                   number you want to change [>07] and the right
                   byte of R0 with the color [>11])
BLWP @>6034        (Execute VDP write to register routine in
                   >6034)
```

To make the screen medium red (>8), you would:

```
LI R0,>0788
BLWP @>6034
```

Bouncing Ball

You may not have realized it, but you have all the elements available to create a simple assembly language game. You've seen how to change character definitions, so you can create almost any kind of figure you want. You've even seen how to change colors, both of the character sets and of the screen.

The following program creates this simple game. It draws a wall around the screen and makes a ball bounce around inside. The program doesn't use sprites—the ball is a redefined character.

The program listing is divided into sections. A general explanation is presented first, then the program segment with an explanation beside each instruction. Since the program runs entirely in assembly language, all character and color tables are located accordingly.

A Simple Game

In this first program section, the initial conditions, such as screen color and variables, are set. The values in the registers will control the movement of the ball. R5 holds the current ball position, R6 the value for the delay loop, R7 the ball's vertical movement (−32 moves it one line up, +32 one line down), and R8 the ball's horizontal movement (1 moves it right, −1 moves it left).

```
7D00     LWPI >70B8       (Load memory area for registers)
7D04     LI R5,300        (Initial position to print the ball)
7D08     LI R6,1000       (Value for the delay loop
7D0C     LI R7,32         (Ball going down initially)
```

7D10	**LI R8,1**	(Ball going down and right initially)
7D14	**LI R0,>0711**	(Load value in R0 for the VWTR routine to change screen color to black)
7D18	**BLWP @>6034**	(Write to VDP register 7)

Next the program defines character 96 for the ball and character 104 for the wall, adding the appropriate definitions (D1 and D2) at the end of the program.

7D1C	**LI R0,>0B00**	(Load position in VDP memory corresponding to character 96)
7D20	**LI R1,D1**	(Load ball definition)
7D24	**LI R2,8**	(Definition is eight bytes long)
7D28	**BLWP @>6028**	(Write the definition to the Character Table in VDP RAM)
7D2C	**LI R0,>0B40**	(Load position in VDP memory corresponding to character 104)
7D30	**LI R1,D2**	(Load wall definition)
7D34	**LI R2,8**	(Load number of bytes to write)
7D38	**BLWP @>6028**	(Write definition to the Character Table in VDP)

The next step is to set the ball and wall colors by changing the colors of character sets 9 and 10. This can be done simultaneously, as you've already seen, by using the VMBM utility.

7D3C	**LI R0,>038C**	(Position in the Color Table corresponding to character set 9)
7D40	**LI R1,CL**	(Position in memory of the two color bytes to be written to the Color Table)
7D44	**LI R2,2**	(Two bytes to be written)
7D48	**BLWP @>6028**	(Write the color bytes to the Color Table)

Now the walls are drawn. This will be done with the VSBW routine and four loops.

7D4C	**CLR R0**	(Initial printing position. Start printing at top-left corner)
7D4E	**LI R1,>6800**	(Load the wall character [104] into R1)

```
7D52      BLWP @>6024     (Print the character)
7D56      INC R0          (Increase printing position)
7D58      CI R0,32        (Has it reached the right side of
                          the screen?)
7D5C      JNE $-10        (If not, return to print a new
                          character, ten bytes back in the
                          program)
7D5E      LI R0,63        (Top wall drawn. Prepare for the
                          right wall. The first printing po-
                          sition will be screen position 63)
7D62      BLWP @>6024     (Print the wall character)
7D66      AI R0,32        (Move the printing position one
                          character down)
7D6A      CI R0,768       (Was it past the bottom-right
                          corner?)
7D6E      JLT $-12        (If not, continue in the printing
                          loop)
7D70      LI R0,32        (If it was, start printing the left
                          wall)
7D74      BLWP @>6024     (Print the wall character)
7D78      AI R0,32        (Move printing position one line
                          down)
7D7C      CI R0,705       (Has bottom left been reached?)
7D80      JLT $-12        (If not, stay in the printing loop)
7D82      LI R0,736       (Start drawing the bottom wall)
7D86      BLWP @>6024     (Print the wall character)
7D8A      INC R0          (Increase printing position)
7D8C      CI R0,768       (Has bottom-right corner been
                          passed?)
7D90      JLT $-10        (If not, stay in printing loop)
```

All that's missing is to put the ball on the screen and set it in motion. The ball will move diagonally in one of four directions. Each time the ball position is updated, the VSBR routine reads a byte from the screen (much like the GCHAR subprogram in BASIC). In this way, the program knows whether the ball has hit a wall to update its direction. If one of the side walls is hit, the ball's horizontal movement will be inverted. If the top or bottom walls are hit, the ball's vertical movement is inverted. This means that the program has to differentiate between the top and bottom walls and the side walls.

One way is to check the ball's position. When the ball hits a wall, if its position is less than screen position 32, the top wall was hit. If the ball's position is greater than 734, the bottom wall was hit. Otherwise, one of the side walls was hit. This is the technique used in the program.

```
7D92   CT   MOV R5,R0         (Move the printing position of the
                              ball, stored in R5, into R0 for the
                              VSBW routine)
7D94        LI R1,>6000       (Load code for ball character)
7D98        BLWP @>6024       (Print ball on the screen)
7D9C        CLR R12           (Start delay loop. Clear R12)
7D9E        INC R12           (Increase the value in R12)
7DA0        C R12,R6          (Is it equal to the delay loop value
                              in R6?)
7DA2        JNE $-4           (If not, stay in the loop)
7DA4        LI R1,>2000       (Delay finished. Load the code of
                              the blank in R1)
7DA8        BLWP @>6024       (Print blank, thus erasing the ball)
7DAC        A R7,R5           (Add ball's vertical movement in
                              R7 to the ball's current position in
                              R5, moving the ball one line up or
                              one line down)
7DAE        A R8,R5           (Update the ball's horizontal po-
                              sition, moving it one character left
                              or right, thus making it move
                              diagonally)
7DB0        MOV R5,R0         (Move the updated print position
                              into R0)
7DB2        BLWP @>602C       (Read the character found in the
                              new ball position to see if a wall
                              was hit)
7DB6        CI R1,>6800       (Has ball hit a wall?)
7DBA        JNE CT            (If not, jump back to CT to print
                              the ball)
7DBC        CI R0,32          (Check to see if it was the top
                              wall hit)
7DC0        JLT NC            (If it was, jump to NC)
7DC2        CI R0,734         (Was the bottom wall hit?)
7DC6        JGT NC            (If it was, jump to NC)
7DC8        NEG R8            (A side wall was hit. Invert the
                              ball's horizontal movement)
```

```
7DCA       A R8,R5      (Update the print position so the
                        ball will not be printed on the
                        wall)
7DCC       JMP CT       (Return to print the ball)
7DCE  NC   NEG R7       (Top or bottom wall hit. Change
                        vertical movement of the ball)
7DD0       A R7,R5      (Update the print position so the
                        ball will not be printed on the
                        wall)
7DD2       JMP CT       (Jump back to print the ball)
```

Finally the character definitions and color bytes are added to the program.

```
7DD4  D1   DATA >183C,>7EFF,>FF7E,>3C18   (Ball definition)
7DDC  D2   DATA >FFFF,>FFFF,>FFFF,>FFFF   (Wall definition)
7DE4  CL   DATA >87A1
```

Execute the program. If you wish to change the speed of the ball, change the value loaded into delay loop register 6, found in address >7D08. Note that the program used the NEG (NEGate) instruction. This simply changes the sign of a value, just as if you had multiplied the value by −1.

Chapter 9

Creating Sprites

Creating Sprites

Another popular feature of the TI-99/4A is its ability to create and display sprites. Sprites are simply large characters that have special properties. One of the nice things about using sprites on the TI is the ease with which you can create and move them. It's much easier, for instance, than designing and moving characters of comparable size.

But you need Extended BASIC in order to use sprites on your computer. That is, if you use BASIC. Fortunately, you can create sprites through assembly language. As long as you have the *Mini Memory* module and its *Line-by-Line Assembler,* you can design, create, and manipulate up to 32 different sprites. These sprites are numbered 0 through 31 and are created by adding certain values to tables in VDP memory.

To create a sprite, you load its position, character code, and color into a table called the *Sprite Attribute List.* Another table, the *Sprite Descriptor Table* is loaded with the sprite's pattern. To a third table, the *Sprite Motion Table,* you add the necessary data if you want the sprite to move automatically (this feature cannot be used if your computer is in bitmap mode). Sprites cannot be used when your computer is in text mode. See Chapter 11 for more information on graphics modes used on the TI.

The Sprite Attribute List

In the Sprite Attribute List, you must include the initial position of the sprite on the screen, the pattern code for the sprite, and its color. Each sprite's entry in the table is four bytes long. In the first byte of the entry you must put the row value of the sprite. Just as with Extended BASIC, you use the (high-resolution graphics) screen for sprites, which consists of 256 dots across and 192 dots down. The dots are called *pixels.* In assembly language, the screen is divided in the same way.

The leftmost column of dots on the screen is column >00. Column >01 is next, then column >02, and so on up to column >FF. The rows are a bit different. The top row is numbered >FF. The second row is >00, the third >01, and so on until the bottom row, the last visible row on the screen, is numbered >BE. Higher numbered rows are off the bottom of the screen.

Decide on the row value of the sprite's initial location and place it in the first byte of the four-byte entry in the Sprite Attribute List. The second byte is loaded with the sprite's column value. The third byte tells the computer the character pattern to use for the sprite. Though you may theoretically use any character number from 0 to 255, it's best to use characters from 128 on up. If you use automatic sprite motion, only characters >80 through >EF (128 through 239) may be used. In the right digit of the fourth byte, you specify the hexadecimal color code (see Appendix D) of the sprite. Just leave the left digit zero for your applications.

Four-Byte Entry in Sprite Attribute List

Byte 1	2	3	4
Row Value	Column Value	Character Code	Zero 1 Color Code

The Sprite Attribute List starts at >0300 in VDP RAM and is divided as follows:

Sprite 0 is at locations >0300 through >0303
Sprite 1 is at locations >0304 through >0307

.
.
.

Sprite 31 is at locations >037C through >037F

To write values in the Sprite Attribute List (which is in VDP RAM), use the VMBW utility, just as you do when you write into any other VDP table. For instance, to make a four-byte entry for sprite 0, you'd enter:

```
     LI R0,>0300          (Location in the VDP Sprite Attribute
                          List where the sprite data will be
                          placed)
     LI R1,DT             (Location in CPU memory where the
                          sprite data will be placed, with label DT)
     LI R2,4              (Number of bytes in entry)
     BLWP @>6028          (Write bytes to the table)
      .
      .
      .
DT   DATA >50A5,>8001     (Data on the sprite to be written to the
                          Sprite Attribute List. The sprite's po-
                          sition will be row >50, column >A5. Its
                          character code is >80 [128] and its color
                          black [>1])
```

As sprites are moved, the corresponding row and column values in the Sprite Attribute List are updated.

Once you've added the sprite information to the table, it's best to disable any unused sprites. You should *always* disable all sprites that have a number higher than the highest numbered sprite you're using. If you are using sprites 0 through 5, for example, you should disable sprites 6 through 31. Disabling sprites is done by placing a value of >D0 in the Y-location (the row byte in the attribute list) of the first unused sprite. In our example, to disable sprites 6–31, you would place the value of >D0 in the row byte (the first byte of the four-byte entry) of sprite number 6.

The Sprite Descriptor Table

You place the sprites' patterns in the Sprite Descriptor Table, in VDP RAM starting at location >0400. Divided into eight-byte blocks which correspond to characters, it's arranged like this:

>0400–>0407 correspond to character >80 (128)
>0408–>040F correspond to character >81 (129)
>0410–>0417 correspond to character >82 (130)

.
.
.

To set the sprite definition, load the character definition into this table at the location of the character you want to use. First write the character definition into memory with the DATA directive and then use the VMBW utility to write the bytes to the Sprite Descriptor Table in VDP RAM. You'll see how to do this in the next example program.

Choosing Sprite Magnification

Sprites can even be magnified, or enlarged. There are four possible magnification modes. In Extended BASIC, this feature is set with the CALL MAGNIFY statement. With a magnification of 1, sprites occupy the same area as a character, in other words eight screen pixels high by eight screen pixels wide. The sprite pattern is defined by eight bytes, just as a standard character pattern. A sprite with magnification of 2 is defined the same as one with magnification 1, except that the sprite pixels have been enlarged so that each sprite pixel occupies four screen pixels. A sprite at magnification 2 covers the same area as a two-character by two-character shape. Magnification factor 3 creates sprites 16 screen pixels high by 16 screen pixels wide. Like magnification 2, it occupies the same area as four characters, but it can have a higher resolution, since you use four character patterns (32 bytes) to define the sprite's pattern. Sprites with magnification 4 are also defined with four character patterns, but each sprite pixel is enlarged so that it's four times as large as in a sprite with magnification 3. Thus, a sprite at magnification 4 covers the same area as a four-character by four-character shape.

In assembly language, the magnification mode of the sprites is set by writing a value into the two least significant (right) bits of VDP register 1. Remember that to write a value into a VDP register, you use the VWTR utility in >6034 (see the section "Changing Screen Color" in the previous chapter). The value you write to the VDP register must be in the right byte of R0 (the left byte of R0 must contain the number of the VDP register you want to change).

For sprites with magnification 1, write the value >E0 into VDP register 1 (see Chapter 11 for a discussion on VDP registers). Again, this is done by placing the magnification value into the right byte of R0 and the number of the VDP register into the left byte of R0.

```
LI R0,>01E0
BLWP @>6034
```

Thus VDP register 1 (as shown by the left byte) is loaded with >E0. For these unmagnified sprites, the two right bits of VDP register 1 must be 0. This means that the byte in VDP register 1 must be (in binary): 111000*00*, which equals 224 in decimal, or E0 in hexadecimal.

Sprites with magnification 2 need to have the value >E1 loaded into VDP register 1:

```
LI R0,>01E1
BLWP @>6034
```

>E1 must be loaded because for these sprites, the two right bits of VDP register 1 must be 0 and 1, in that order from left to right. In binary, the corresponding value then is 111000*01*, or 225 in decimal, which is E1 in hexadecimal.

Magnification 3 requires >E2 loaded into VDP register 1:

```
LI R0,>01E2
BLWP @>6034
```

These sprites have to have the two last bits in VDP register 1 to be 1 and 0. The binary value would be 111000*10* (226 in decimal), which is E2 in hex.

Sprites with magnification of factor 4 need >E3 loaded into VDP register 1:

```
LI R0,>01E3
BLWP @>6034
```

The value written into VDP register 1 must have the two last bits set, or 1. In binary, this is 111000*11*. That's 227 in decimal, or E3 in hexadecimal.

When you change the value in VDP's register 1, also place the value in CPU RAM location >83DA. If you don't, and execute a KSCAN, the register will be reset to its original value. By writing the value into >83DA, the computer updates the value in VDP register 1 whenever necessary by placing the value found in >83DA into that register. (This wasn't done in the sample programs here, because the KSCAN routine was not used after the value of VDP register 1 had been changed. If it was, it was reset to a new value after the KSCAN.)

Here's how to write a value into both VDP register 1 and address >83DA:

```
LI R0,>01E1          (Value to be written into VDP register 1
                     [>E1])
BLWP @>6034          (Write >E1 to VDP register 1)
SWPB R0              (Place >E1 into the left byte of R0)
MOVB R0, @>83DA      (Move the left byte of R0 [>E1] into ad-
                     dress >83DA)
```

Static Sprite

Before we go on to see how a sprite can be moved, let's look at a program which simply puts a sprite on the screen. The program places a sprite with magnification 1 on the screen and waits for you to press a key, which enlarges the sprite.

Sprite—Magnification 1

The first step is to load the sprite definition into the Sprite Descriptor Table:

```
7D00     LWPI >70B8     (Load memory area for registers)
7D04     LI R0,>0400    (Use character 128 [>80] for the
                        sprite definition, so use the first
                        entry in the table [>0400–>0407])
```

```
7D08        LI R1,DF           (Load position in CPU RAM
                               where the definition will be, with
                               label DF)
7D0C        LI R2,8            (Definition is eight bytes long)
7D10        BLWP @>6028        (Write definition bytes to the
                               table)
```

Then complete the Sprite Attribute List with the sprite data:

```
7D14        LI R0,>0300        (Position in the Sprite Attribute
                               List where the sprite data will be
                               written. Use sprite 0, from >0300
                               to >0303)
7D18        LI R1,DT           (Position in CPU RAM where the
                               sprite data can be found)
7D1C        LI R2,5            (Five bytes to write; the first four
                               bytes describe the sprite. The fifth
                               byte will be >D0 and disables
                               sprites 1–31)
7D20        BLWP @>6028        (Write data to the Sprite Attribute
                               List)
```

The sprite is now displayed on the screen. Next you'll create a loop to read the keyboard until any key is pressed. When this happens, the sprite is enlarged by writing the new value into VDP register 1. Then program execution is stopped.

```
7D24        CLR @>8374         (Clear byte at >8374. Standard
                               keyboard scan)
7D28   LP   BLWP @>6020        (KSCAN loops start)
7D2C        MOVB @>837C,R1     (Move status byte into R1)
7D30        COC @BT,R1         (Compare Corresponding Ones of
                               both bytes)
7D34        JNE LP             (If bytes are not equal, no key has
                               been pressed. Stay in loop)
7D36        LI R0,>01E1        (Key pressed. Change magnifica-
                               tion factor to 2 by writing >E1
                               into VDP register 1)
7D3A        BLWP @>6034        (Write new value to VDP register
                               1)
7D3E        JMP $              (Sprite magnified. Stop program
                               with an endless loop)
```

Finally, add the necessary data:

7D40	BT	DATA >2000	(Comparison value for the KSCAN loop)
7D41	DF	DATA >FF81,>8181,>8181,>81FF	(Character definition for the sprite)
7D4A	DT	DATA >6080,>8004,>D000	(Data for the Sprite Attribute List. >60 [96] is the sprite's row number. The next >80 is the sprite's character definition in the Sprite Descriptor Table [128]. >04 is the sprite color. >D0 is the value used to disable the remaining sprites.)

End the program and execute it with EASY BUG. A blue square the size of a character will appear on the screen. Press any key and the box will be magnified. The program then stops.

(You didn't have to specify that the sprite should be in magnification 1, since that's the default magnification mode.)

The next program does much the same thing, but the sprite will be set initially to magnification 3. When a key is pressed, the magnification mode will be 4. The sprite's definition, then, has to be of four characters.

Sprite—Magnification 3

First of all, the program adds the sprite definition to the Sprite Descriptor Table. The sprite's character definition uses characters 132 through 135 (>84 through >87). The starting position in the Sprite Descriptor Table for character >84 is address >0420.

7D00	LWPI >70B8	(Load memory area for registers)
7D04	LI R0,>0420	(Starting position in the Sprite Descriptor Table of the first of the four characters to be defined)
7D08	LI R1,DF	(The definition of the four characters is in CPU RAM, starting at label DF)
7D0C	LI R2,32	(Number of bytes written. Each definition is eight bytes.)
7D10	BLWP @>6028	(Write bytes to VDP table)

Next you must add the sprite information to the Sprite Attribute List. Sprite 0 is used, and its entry in the table is from >0300 to >0303.

```
7D14        LI R0,>0300        (Position in the Sprite Attribute
                               List for sprite 0's entry)
7D18        LI R1,DT           (Position in CPU RAM where the
                               sprite data is)
7D1C        LI R2,5            (Five-byte data: four bytes for
                               sprite description and one to dis-
                               able the remaining sprites)
7D20        BLWP @>6028        (Write bytes to table)
```

Now add the sprite magnification (3 to start with):

```
7D24        LI R0,>01E2        (Write >E2 into VDP R1 to in-
                               dicate magnification factor of 3)
7D28        BLWP @>6034        (Write to VDP register)
```

The sprite is now on the screen. To make the program wait for a key to be pressed and then enlarge the sprite to magnification 4, you need to enter:

```
7D2C        CLR @>8374         (Standard keyboard scan)
7D30  LP    BLWP @>6020        (Branch to KSCAN routine)
7D34        MOVB @>837C,R1     (Move status byte into R1)
7D38        COC @BT,R1         (Compare Ones Corresponding of
                               both bytes)
7D3C        JNE LP             (If not equal, stay in the KSCAN
                               loop)
7D3E        LI R0,>01E3        (Key pressed. To set magnification
                               factor 4, write >E3 into VDP reg-
                               ister 1)
7D42        BLWP @>6034        (Write to VDP register)
7D46        JMP $              (Stop program execution with an
                               endless loop)
```

Finally, add the data values for sprite creation and the KSCAN loop:

```
7D48  BT    DATA >2000         (Comparison value for the
                               KSCAN loop)
```

```
7D4A  DF  DATA >0300,>1020,>4040,>8080 (Sprite definition
                                        [four characters]. Top left charac-
                                        ter defined first. Bottom left, top
                                        right, and bottom right follow)
7D52      DATA >8080,>4040,>2010,>CC03
7D5A      DATA >C030,>0804,>0202,>0101
7D62      DATA >0101,>0202,>0408,>30C0
7D6A  DT  DATA >6080,8401,>D000 (Same values as the previous
                                        program, except the second word
                                        includes >84, which signifies that
                                        this version uses characters 132
                                        onwards, and >01, to indicate
                                        black color)
```

End the program and run it. The large unmagnified sprite appears on the screen. Press any key and it's magnified.

Sprite Motion

You can move the sprites in two ways. The first, which you'll see in this section, is to move the sprite pixel by pixel. All you have to do, once you have set the sprite on the screen, is change the row and/or column value of the sprite by writing the new values into the Sprite Attribute List. The program below places a sprite on the screen and moves it to the right by constantly increasing the column byte value in the Sprite Attribute List.

Pixel by Pixel

First of all, the sprite definition is loaded into memory:

```
7D00      LWPI >70B8        (Load memory area for registers)
7D04      LI R0,>0400       (Use character >80 [128] for the
                            sprite, with entry at >0400 in the
                            Sprite Descriptor Table)
7D08      LI R1,DF          (Sprite definition is in CPU RAM
                            address labeled DF)
7D0C      LI R2,8           (Definition is eight bytes long)
7D10      BLWP @>6028       (Write definition to Sprite Descrip-
                            tor Table)
```

Sprites with magnification 2 will be used:

7D14		**LI R0,>01E1**	(Load value of >E1 to write it to VDP register 2 to select magnification 2)
7D18		**BLWP @>6034**	(Write new value for VDP register 1 with VWTR utility)

The sprite's initial screen position is set by writing the corresponding data to the Sprite Attribute List:

7D1C		**LI R0,>0300**	(Use sprite 0, position >0300 in the table)
7D20		**LI R1,DT**	(Load sprite data from CPU RAM address labeled DT)
7D24		**LI R2,5**	(Five bytes to write to table)
7D28		**BLWP @>6028**	(Write bytes to table)

Increase the column value of the sprite, keeping track of it in a register, in loops from 0 to 255, constantly updating the column value in the Sprite Attribute List. Remember that the column value for sprite 0 is found in VDP byte >0301.

7D2C		**LI R0,>0301**	(Load position of the sprite column byte in the table)
7D30	**RS**	**CLR R1**	(Prepare R1 to keep track of the sprite's current column position)
7D32	**LP**	**AI R1,>0100**	(Move sprite column one position [pixel] right)
7D36		**BLWP @>6024**	(Write new column value to the Sprite Attribute List)
7D3A		**CI R1,>FF00**	(Has the last column been reached?)
7D3E		**JEQ RS**	(If it has, jump to RS to reset column value)
7D40		**LI R15,1000**	(Load value for delay loop)
7D44		**DEC R15**	(Decrease delay loop value)
7D46		**JNE $-2**	(If not zero, jump to >7D44)
7D48		**JMP LP**	(Delay loop finished. Return to loop)

Finally, add the sprite definition and the data for the Sprite Attribute List:

```
7D4A  DF  DATA >FF81,>8181,>8181,>81FF (Sprite definition)
7D52  DT  DATA >5000,>8001,>D000 (Sprite data. Put initial
                                  sprite in row >50, column >00;
                                  use character >80 for the sprite;
                                  and make the sprite black [>01].
                                  Disable remaining sprites [>D0].)
```

End the program and run it. The sprite displays on the screen and sails across it. If you want to change the speed of the sprite, use AORG to get to the position in memory where the delay loop was added (>7D40) and change the value loaded in R15 to whatever number you wish. If you load a 1 into R15, the sprite will move so quickly that you'll see what looks like several sprites blinking on the screen. There's only one sprite—it's just moving so fast that you get the illusion several are slowly moving from right to left.

Though it was included in the program listing, you really don't need to check whether the column value in the left byte of R1 is equal to >FF to reset it. This happens automatically. If a register is loaded with >FF (255) and you add 1, the value rolls over to 0. Instructions in addresses >7D3A and >7D3E could be left out of the program because of this.

Using the Sprite Motion Table

Another way to move sprites is to let the computer do it. In order to use this feature, you must use the Sprite Motion Table.

Before automatically moving sprites, interrupts must be enabled with the LIMI 2 instruction. Remember to disable them again with the LIMI 0 instruction before accessing any VDP RAM. Accessing VDP RAM with the interrupts enabled might bring you disastrous results, as values in VDP are changed. It's best to enable and disable interrupts quickly in a frequently executed loop, like this:

```
   .
   .
   .
LP LIMI 2
   LIMI 0
```

```
.
.
.
JMP LP
.
.
.
```

If you do this, you don't need to worry whether you remembered to enable or disable interrupts at any point in the program. For more information, see Chapter 6.

Before accessing the Motion Table, you also must tell the computer how many sprites will be moved. If sprites 3 and 7 are going to move, for instance, you'll have to allow movement of the *first eight* sprites, 0 through 7. To do this, you'd place an 8 at address >837A:

```
LI R5,>0800
MOVB R5,@>837A
```

The above program segment writes a value of 8 into memory address >837A, thus allowing the automatic motion of sprites 0 through 7. If you'd entered LI R5,>0600 instead, it would have allowed the automatic motion of only sprites 0–5. (Note: The number placed in >837A indicates the *total* number of sprites that will be able to move automatically.)

Once you've allowed VDP interrupts and placed the number of moving sprites in location >837A, you can add the sprite data to the Motion Table.

The Motion Table starts at VDP address >0780, and each sprite's entry takes four bytes.

Sprite 0:	>0780	through >0783
Sprite 1:	>0784	through >0787
Sprite 2:	>0788	through >078B
Sprite 3	>078C	through >078F
Sprite 4:	>0790	through >0793
Sprite 5:	>0794	through >0797
Sprite 6:	>0798	through >079B
Sprite 7:	>079C	through >079F

.
.
.

In each sprite's entry, the first byte indicates the vertical motion of the sprite and the second, the horizontal motion. The third and fourth bytes are used by the computer and should be zero.

Loading a value from >00 to >7F (0 to 127 decimal) in either the first or second bytes gives a *positive* velocity. Values from >FF to >80 (255 to 128 decimal) are *negative* velocities, where

>FF = −1
>FE = −2
.
.
.
>80 = −127

In other words, a value from >00 to >7F in the first byte moves the sprite down and a value from >FF to >80 moves it up. A value from >00 to >7F in the second byte moves the sprite to the right, and a value from >FF to >80 moves the sprite left.

Once you've loaded this information into the Motion Table, your sprites will move automatically. You don't need to worry about them again.

Automatic Sprites

The following program moves a sprite across the screen, just like in the "Pixel by Pixel" program, except that the sprite moves under computer control. The row and column values of the moving sprites in the Sprite Attribute List will be updated by the computer.

First of all, the program sets the sprite definition:

7D00	**LWPI >70B8**	(Load memory area for registers)
7D04	**LI R0,>0400**	(Position in the Sprite Descriptor Table where sprite definition is written)
7D08	**LI R1,DF**	(Position in CPU RAM where the definition will be loaded with label DE)
7D0C	**LI R2,8**	(Definition is eight bytes long)
7D10	**BLWP @>6028**	(Write definition to the table)

Next the sprite's magnification is set to 2:

7D14	**LI R0,>01E1**	(Prepare to write >E1 to VDP register 1)
7D18	**BLWP @>6034**	(Write to VDP register 1)

Then the sprite attributes are set in the Sprite Attribute List:

```
7D1C         LI R0,>0300        (Place in table for sprite 0)
7D20         LI R1,DT           (Position in CPU RAM where the
                                sprite data will be loaded with la-
                                bel DT)
7D24         LI R2,5            (Five bytes to write)
7D28         BLWP @>6028        (Write data to table)
```

The number of automatically moving sprites is placed at location >837A. To allow sprite 0 to move under computer control, place a 1 in that address.

```
7D2C         LI R1,>0100        (Load left byte of R1 with value to
                                be written)
7D30         MOVB R1,@>837A     (Move the left byte of R1 into
                                >837A)
```

And add the velocity data to the Motion Table:

```
7D34         LI R0,>0780        (Entry in table for sprite 0)
7D38         LI R1,VD           (Motion data is in CPU RAM, la-
                                beled VD)
7D3C         LI R2,4            (Four bytes of data to be written)
7D40         BLWP @>6028        (Write data to table)
```

The sprites will be moving as soon as interrupts are enabled. Program execution is stopped with a loop.

```
7D44    LP   LIMI 2             (Enable VDP interrupts to allow
                                automatic sprite motion)
7D48         LIMI 0             (Disable VDP interrupts)
7D4C         JMP LP             (Jump back to "end of program"
                                loop)
```

Finally, add the data values for the program:

```
7D4E    DF   DATA >FF81,>8181,>8181,>81FF (Sprite definition)
7D56    DT   DATA >6000,>8001,>D000 (Sprite data. Initial sprite
                                location is row >60, column >00.
                                Character >80 is used, and the
                                sprite is black [>01]. Remaining
                                sprites are disabled [>D0])
```

```
7D5C   VD   DATA >007F,>0000 (Motion data. Vertical motion
                             is zero [>00]. Horizontal motion is
                             maximum positive [>7F], and the
                             two remaining bytes are left as
                             zero)
```

End and run the program. The sprite moves under computer control until you press FCTN = (QUIT) to stop the program (QUIT was enabled by the VDP interrupts). Note that with automatic motion, sprites cannot be moved faster than with Extended BASIC. The values of sprite velocity in both Extended BASIC and assembly language range from −127 to 127.

Checking for Coincidences

To check for a coincidence between two sprites (another term for the same thing is *collision*), you simply read the row and column values of both sprites from the Sprite Attribute List and compare them.

Don't expect to compare the value pairs and find them to be equal. A sprite colliding with another would then be detected only if the two top left-hand side pixels coincide *exactly*. The best way to detect collisions between sprites is to read the row and column values of both sprites and subtract them, taking the absolute value of the answer. This will tell you how many pixels there are between each row and column coordinates. If both these values are less than nine pixels, a crash has occurred. Otherwise, no collision happened.

The number of pixels between two points should always be positive, so you take the *absolute value* to account for the cases when the row of the first sprite is less than the row of the second sprite.

To read the row and column values of the sprites from the Sprite Attribute List, you can use either the VSBR or VMBR utilities.

Collisions

This program sets two sprites moving in different directions across the screen. The program continues until the sprites collide. The sprite motion and the program then stop.

First of all, you need to load the character definitions for both sprites into the Sprite Descriptor Table:

```
7D00    LWPI >70B8          (Load memory area for registers)
7D04    LI R0,>0400         (Location in the table to start writ-
                            ing sprite definitions)
7D08    LI R1,DF            (Position of the definitions in CPU
                            memory)
7D0C    LI R2,16            (Each definition is eight bytes
                            long. Two definitions, or 16 bytes,
                            to be written)
7D10    BLWP @>6028         (Write definitions to the table, de-
                            fining characters >80 and >81)
```

Set a magnification factor of 2 for both sprites:

```
7D14    LI R0,>01E1         (Load VDP register number and
                            the value to be written to it in R0)
7D18    BLWP @>6034         (VDP Write to Register)
```

Add the number of moving sprites at location >837A:

```
7D1C    LI R7,>0200         (Sprites 0 and 1 will move. Pre-
                            pare to write a 2 in location
                            >837A)
7D20    MOVB R7,@>837A      (Move number of moving sprites
                            to >837A)
```

Load the data for both sprites in the Sprite Attribute List:

```
7D24    LI R0,>0300         (Position for sprite 0 in the table)
7D28    LI R1,DT            (Position of sprite data in CPU
                            memory, label DT)
7D2C    LI R2,9             (Nine bytes to move: four bytes
                            for each sprite and one byte to
                            disable the remaining sprites)
7D30    BLWP @>6028         (Write sprite data to the Sprite
                            Attribute List)
```

Place the data for both sprites in the Motion Table:

```
7D34    LI R0,>0780         (Position in the table for sprite 0)
7D38    LI R1,MD            (Motion data is at CPU RAM ad-
                            dress labeled MD)
```

```
7D3C          LI R2,8          (Eight bytes of motion data to be
                               written, four bytes for each sprite)
7D40          BLWP @>6028      (Write data to the table)
```

The sprites are now on the screen. Start a loop to enable and disable VDP interrupts, read the row and column values of both sprites, compare them and branch accordingly:

```
7D44    LP    LIMI 2           (Enable VDP Interrupts)
7D48          LIMI 0           (Disable VDP Interrupts)
7D4C          LI R0,>0300      (Load position of the row value of
                               sprite 0 in the Sprite Attribute List
                               in VDP RAM)
7D50          CLR R1           (Prepare R1 to receive the row
                               value of sprite 0)
7D52          CLR R2           (Prepare R2 to receive the value
                               read in R1)
7D54          BLWP @>602C      (Read the row value of sprite 0
                               into the left byte of R1)
7D58          MOVB R1,R2       (Move it into the left byte of R1)
7D5A          SWPB R2          (Place the row value of sprite 0 in
                               the right byte of R2)
7D5C          LI R0,>0304      (Load position of the row value of
                               sprite 1 in the Sprite Attribute
                               List)
7D60          BLWP @>602C      (Read the row value of sprite 1
                               into the left byte of R1)
7D64          SWPB R1          (Place it in the right byte of R1)
7D66          S R1,R2          (Subtract the row values. The an-
                               swer is placed in the right byte of
                               R2)
7D68          ABS R2           (Make it the absolute value)
7D6A          CI R2,>0008      (Compare it to eight)
7D6E          JGT LP           (If the result is greater than eight,
                               there has been no crash. Return to
                               loop LP)
7D70          LI R0,>0301      (Possible crash. Now check the
                               sprite columns. Load position of
                               the column byte of sprite 0 in
                               VDP RAM)
7D74          CLR R1           (Prepare R1 to receive the column
                               value)
```

```
7D76    CLR R2          (Prepare R2 to receive the value
                        of R1)
7D78    BLWP @>602C     (Read the column byte into the
                        left byte of R1)
7D7C    MOVB R1,R2      (Store it in the left byte of R2)
7D7E    SWPB R2         (Move it to the right byte of R2)
7D80    LI R0,>0305     (Load position of the column byte
                        of sprite 1)
7D84    BLWP @>602C     (Read the column value)
7D88    SWPB R1         (Place it in the right byte of R1)
7D8A    S R1,R2         (Subtract the column values)
7D8C    ABS R2          (Make it the absolute value)
7D8E    CI R2,>0008     (Compare it to eight)
7D92    JGT LP          (If it's greater than eight, no crash
                        occurred. Return to loop LP)
```

When execution leaves the loop and continues past >7D92, it means that the row and column values of each sprite are within eight pixels of each other, and a collision *has* occurred. The sprite motion is then stopped by writing zeros into the Sprite Motion Table, and following that, the program itself stops.

```
7D94    LI R0,>0780     (Load position in the table for
                        sprite 0)
7D98    LI R1,MS        (Load motion data [all 0's] to stop
                        sprite motion)
7D9C    LI R2,8         (Eight bytes of motion data to
                        write)
7DA0    BLWP @>6028     (Write the data to VDP RAM)
7DA4    JMP $           (Stop program execution with an
                        endless loop)
```

Finally, add the necessary data values:

```
7DA6  DF  DATA >FF81,>8181,>8181,>81FF (Definition of
                        sprite 0)
7DAE      DATA >1824,>4281,>8142,>2418 (Definition of
                        sprite 1)
7DB6  DT  DATA >3030,>8001,>C8C8,>8101,>D000 (Data for
                        both sprites)
```

```
7DC0  MD  DATA >B050,>0000,>7030,>0000 (Data to set the
                                       sprites moving)
7DC8  MS  DATA 0,0,0,0                 (Data to stop the sprites once
                                       again)
```

End and run the program. Note that even if the sprites are moving at a relatively high speed, the crash is detected accurately and the sprite motion is stopped immediately (because VDP interrupts are disabled). Coincidence checking can be quite tricky. In the program above, collisions are detected only if the two top left-hand corners of the sprites are within eight pixels of each other, in any direction.

Vanishing Sprites

You can even make sprites disappear and then later reappear. This can be quite handy when programming games. To make a sprite disappear, you can make it transparent or set it to the same color as the screen. Or you can position it at the unused memory area of the screen by assigning it a row coordinate greater than >BE. The sprite will then be in the screen area not visible; in other words, just off the bottom of the screen.

To make a sprite vanish, just change its row value (the first byte of the sprite's entry in the Sprite Attribute List) to a value a little greater than >BE, the bottom pixel of the visible screen. You can use >C0, two bytes greater than >BE.

Poof!

This program puts a sprite on the screen and waits for you to press a key. When the key is detected, the row value of the sprite is changed to a position off the screen, creating the illusion that the sprite has vanished.

First of all, load the Sprite Descriptor Table:

```
7D00        LWPI >70B8        (Memory area for registers)
7D04        LI R0,>0400       (Position in the table where the
                              definition will be written)
7D08        LI R1,DF          (Position of the sprite definition in
                              CPU RAM)
7D0C        LI R2,8           (Definition is eight bytes long)
7D10        BLWP @>6028       (Write the definition to the table)
```

Write the sprite data to the Sprite Attribute List:

```
7D14        LI R0,>0300        (Entry for sprite 0 in the Sprite
                               Attribute List)
7D18        LI R1,DT           (Sprite data in CPU memory ad-
                               dress labeled DT)
7D1C        LI R2,5            (Five bytes of data)
7D20        BLWP @>6028        (Write sprite data to the table)
```

The sprite is now on the screen. Start a loop to read the keyboard and wait for a key to be pressed. When this happens, the row number of the sprite is changed to >C0, and the sprite seems to disappear:

```
7D24        CLR @>8374         (Standard keyboard scan)
7D28   LP   BLWP @>6020        (Branch to read the keyboard)
7D2C        MOVB @>837C,R1     (Move status byte into R1)
7D30        COC @BT,R1         (COC of the byte at address BT
                               and the byte in R1)
7D34        JNE LP             (If not equal, stay scanning the
                               keyboard)
7D36        LI R0,>0300        (Key pressed. Load address of the
                               row byte in VDP RAM, to change
                               it)
7D3A        LI R1,>C000        (Load the new row value [>C0,
                               off the screen] into the left byte of
                               R1)
7D3E        BLWP @>6024        (Write the new value in the table)
7D42        JMP $              (Stop program execution with an
                               endless loop)
```

Add the DATA statements for the KSCAN loop and sprite creation:

```
7D44   BT   DATA >2000                        (Comparison
                                              value for the
                                              KSCAN loop)
7D46   DF   DATA >FF81,>FF81,>FF81,>FF81      (Sprite pattern)
7D4E   DT   DATA >6080,>8001,>D000            (Sprite data)
```

Deleting All Sprites in Assembly Language

To make all the sprites displayed disappear at once, just as done with the CALL DELSPRITE(ALL) statement in Extended BASIC, all you have to do is disable sprite 0.

This will disable the other 31 sprites at the same time. To disable sprite 0, write the value >D0 into the Sprite Attribute List at the row position of the sprite:

```
LI R0,>0300
LI R1,>D000
BLWP @>6024
```

When these three instructions are executed, all sprites will vanish from the screen. Easy, wasn't it?

Equivalents to BASIC

There are, of course, statements and commands in Extended BASIC which access sprites. Those commands, and their assembly language equivalents, are:

Extended BASIC	**Assembly Language**
CALL CHAR	Write the sprite definition into the Sprite Descriptor Table to the appropriate character.
CALL SPRITE	Write the corresponding sprite data into the Sprite Attribute List (does not include automatic motion).
CALL MOTION	Enable interrupts, set number of moving sprites in >837A, and write the motion values into the motion table.
CALL PATTERN	Change the third byte of the sprite entry (the character code byte) in the Sprite Attribute List to the new character with the new definition.
CALL LOCATE	Write the new row and column values into bytes 1 and 2 of the sprite entry in the Sprite Attribute List.
CALL COLOR	Write the new color for the sprite into the right digit of the fourth byte of the sprite entry in the Sprite Attribute List. Leave zero in the left byte.
CALL COINC	Read the row and column values of the sprites from the Sprite Attribute List and compare them accordingly.

CALL DELSPRITE (#X)	Change the row coordinate of the sprite to a value greater than the bottom visible row of the screen.
CALL DELSPRITE (ALL)	Write D0 into the row position of sprite 0 in the Sprite Attribute List.
CALL MAGNIFY	Set the magnification mode by changing the value loaded in VDP register 1.

Calling from BASIC

All you've learned about sprites in assembly language can also be applied when your program is called from BASIC. All the sprite tables are in the same memory areas when your assembly language program is called from BASIC. The only thing you have to be careful with is the character definitions. If you write the definitions of the sprites starting with character 128 (>80), you'll also be redefining character 32. It's best to use characters 129 and up, noting what characters are simultaneously redefined in BASIC.

That's one of the reasons why it's usually best to write programs which use sprites entirely in assembly language.

After All of This

You should be prepared for some possible strange effects when you start experimenting with sprites. These effects will most likely be caused by some missing value, or a value put in the wrong memory address. Test patterns, colors, coincidences, and so on before you start to write a program using sprites; this will help you avoid problems when you're actually writing your own assembly language sprite programs.

Chapter 10

Generating Sounds

Generating Sounds

The TI-99/4A's sound capabilities are impressive. Like the other features of your computer, sound can be set and used through assembly language.

The TI's sound chip can generate up to three tones and one noise simultaneously. When accessed through assembly language, the tones have the same frequencies as in the BASIC CALL SOUND subroutine and extend up to 55938 Hz. You must also set the volume and duration of the sound as you normally do in BASIC.

Tables

To generate a sound, you create a table of data which must be located in VDP RAM. A convenient location for the table is address >1000. This data contains all necessary information for the sounds to be played by the computer. The length of this table depends on how many sounds will be played. Once you've created the table, you can start sound generation.

At CPU address >83CC, you tell the computer where the sound data can be found in VDP RAM. You then inform the TI that the data is located in VDP RAM by setting the rightmost bit (bit seven) of byte >83FD to 1. The sound is activated by placing a value of >01 at address >83CE.

Once the sound generation has begun, to allow the sound to be heard you have to quickly enable and disable the VDP interrupts with the usual instructions:

```
LIMI 2
LIMI 0
```

This can be done in a frequently executed loop or each time a sound is generated. Keep in mind that VDP interrupts must be disabled with the LIMI 0 instruction when you're writing values to VDP RAM.

This procedure will generate one or more sounds, providing that the table of data describing the sound(s) is found in VDP RAM. This table can be POKEd into memory from BASIC. It's simpler than using CPU RAM to write the data lists and then move them to VDP RAM.

Creating Sound Data Tables

When creating sound data tables, you'll use three bytes for a single tone and two bytes for noise. At the beginning of each entry, specify how many bytes must be read to describe the sound (for simultaneous tones and noise, add the bytes needed to describe each tone and the noise), and at the end of each entry, a byte to indicate the general duration of the sound. The duration byte is not counted in the initial "counter" byte.

Tones are played using three *generators*, numbered 1, 2, and 3. Noises, strangely enough, are created by a *noise generator* (also referred to as generator 4).

The first byte of each sound entry is loaded with the number of bytes to be used to describe the sound (the duration byte is not included). A tone requires three bytes; a noise, two. For example, an entry to play a single tone could be:

```
DATA >038C,>1F91,>1E00
```

The first byte, >03, indicates that three bytes will be loaded into the sound processor. Those bytes are >8C, >1F, and >91. The first two specify the frequency and the third specifies the volume. >1E is the last byte of the entry and is the duration of the sound (remember, it's not counted in the counter byte).

Figuring out the values of the second, third, fourth, and fifth bytes can be a lot of work. You'll be setting and resetting bits and doing binary to hexadecimal conversions. The BASIC program below translates the values used to create a sound in TI BASIC to a hexadecimal DATA value. The program only changes single tones or noises. To combine these tones and noises, refer to the next section.

Creating DATA—A Utility Program

The following BASIC program asks you for the duration, frequency, and volume of a tone or noise, as well as the generator number (1, 2, or 3 for tones; 4 for noise). It then calculates the appropriate values in hexadecimal to be POKEd into memory.

```
100 CALL CLEAR
110 PRINT " *SOUND DATA TABLE CREATOR* ":::::
120 Q$="0123456789ABCDEF"
130 INPUT "GENERATOR # ?":GN
140 INPUT "DURATION ?":DUR
150 INPUT "FREQUENCY ?":FREQ
160 INPUT "VOLUME ?":VOL
170 PRINT :::
180 IF DUR >17 THEN 200
```

```
190 DUR=17
200 REM DURATION
210 DUR=INT((DUR*255)/4250)
220 CONV=DUR
230 GOSUB 540
240 DUR$=SEG$(HX$,3,2)
250 IF FREQ >-1 THEN 370
260 REM NOISE FREQUENCY
270 FR=ABS(FREQ)-1
280 FR$="E"&STR$(FR)
290 REM NOISE VOLUME
300 VOL=INT(VOL/2)
310 CONV=VOL
320 GOSUB 540
330 VOL$="F"&SEG$(HX$,4,1)
340 PRINT "DATA > 02";FR$;",>";VOL$;DUR$:::
350 GOTO 470
360 REM TONE FREQUENCY
370 FR=INT((111860.8/FREQ)+.5)
380 CONV=FR
390 GOSUB 540
400 FR$=SEG$(Q$,GN*2+7,1)&SEG$(HX$,4,1)&SEG$(HX$,2,2)
410 REM TONE VOLUME
420 VOL=INT(VOL/2)
430 CONV=VOL
440 GOSUB 540
450 VOL$=SEG$(Q$,GN*2+8,1)&SEG$(HX$,4,1)
460 PRINT "DATA >03";SEG$(FR$,1,1)&SEG$(FR$,2,1);",>";SEG$(
    FR$,3,2);VOL$;",>";DUR$;"00":::
470 PRINT :::"ANOTHER SOUND ?(Y/N)"
480 CALL KEY(0,K,S)
490 IF K=89 THEN 100
500 IF K=78 THEN 520
510 GOTO 480
520 CALL CLEAR
530 END
540 REM DECIMAL TO HEX
550 AY=INT(CONV)/16
560 BY=INT(AY)/16
570 CY=INT(BY)/16
580 DY=INT(CY)/16
590 AP=(AY-INT(AY))*16
600 BP=(BY-INT(BY))*16
610 CP=(CY-INT(CY))*16
620 DP=(DY-INT(CY))*16
```

```
630 HX$=SEG$(Q$,DP+1,1)&SEG$(Q$,CP+1,1)&SEG$(Q$,BP+1,1
   )&SEG$(Q$,AP+1,1)
640 RETURN
```

Running the Program

Run the program. (You don't need the *Mini Memory* cartridge plugged in.) You'll first be asked what generator the sound should play on. If you're creating several tones, enter 1, then 2, then 3, and repeat. To create a noise, enter 4. Next enter the duration, or length, of the sound (0–4250). Then type in the frequency (from 110 to 55938) in hertz, and the volume (0–30). When you press ENTER, the DATA statement which will create that sound appears. The program does not check for bad values, so make sure you're typing in values within the appropriate range at each prompt.

Let's take a look at an example. You want to create a single tone, two seconds long, with a frequency of 185 and a volume of 4. You would then enter:

```
GENERATOR #? 1
DURATION ? 2000
FREQUENCY ? 185
VOLUME ? 4
```

and this line will appear on the screen:

```
DATA >038D,>2592,>7800
```

When these values are added with the DATA directive to an assembly language program, the sound will execute.

How about another example? For a noise of frequency −5, duration 1000, and volume 2, give these inputs:

```
GENERATOR #? 4
DURATION ? 1000
FREQUENCY ? -5
VOLUME ? 2
```

and the program will respond with:

```
DATA >02E4,>F13C
```

To create several tones, or tones and noise together, find the DATA values for each tone or noise separately and join them. The only byte you'll have to change is the first byte, specifying the number of bytes to be read.

Multiple Tones

If you wanted to create two simultaneous tones and a noise, with a duration of three seconds—tone 1 of frequency 440, and volume 4; tone 2 of frequency 880 and volume 8; and a noise of frequency −3 and volume 2—you'd enter:

```
GENERATOR #? 1
DURATION ? 3000
FREQUENCY ? 440
VOLUME ? 4
```

after which you'd see:

```
DATA >038E,>0F92,>B400
ANOTHER SOUND ? (Y/N)
```

Pressing the *Y* key would let you enter another sound.

```
GENERATOR #? 2
DURATION ? 3000
FREQUENCY ? 880
VOLUME ? 8
```

```
DATA >03AF,>07B4,>B400
ANOTHER SOUND ? (Y/N) Y
```

```
GENERATOR #? 4
DURATION ? 3000
FREQUENCY ? −3
VOLUME ? 2
```

and the final DATA would be:

DATA >02E2,>F1B4

Now you have to join the DATA values. The first byte (number of bytes to be read) is found by adding the first bytes of each sound to form a new counter byte. That would be >08 (>03+>03+>02). This is done to load the sounds into different generators, and thus allow the sounds to play simultaneously.

The frequency and volume bytes of each sound are written sequentially, but the duration byte is excluded. Remember that each tone uses three bytes for frequency and sound, while noise uses only two. The appropriate bits, in order, are: >8E, >0F, >92, >AF, >07, >B4, >E2, and >F1. Add a general duration byte to the end of the entry. Use >B4 for this example, since it signifies a length of 3000. The complete DATA line for the two tones and noise would then be:

```
DATA >088E,>0F92,>AF07,>B4E2,>F1B4
```

If necessary, the duration byte can be padded with two trailing zeros (>B4 becomes >B400, for instance) if the memory word is the last in the entry. Otherwise, just complete the word with the first byte from the next entry. Don't leave zero bytes (>00) in the middle of a sound entry. In other words, you must not leave unused bytes between tones. For example, in the DATA directives below, which play two sounds, one after another:

```
DATA >038E,>0F92,>7800
DATA >038F,>0794,>3600
```

the >00 after the >78 is an unused byte, so the >03 of the following entry can occupy that position. The above *should* have been written as:

```
DATA >038E,>0F92,>7803
DATA >8F07,>9436 ... (and the sound list continues)
```

Volume Changes

You can also change the volume of a specified generator without changing the frequency. The volume of a generator is specified by one byte. The left digit of the byte tells you what generator the volume is describing:

>9—generator 1
>B—generator 2
>D—generator 3
>F—noise generator (4)

The right digit of the byte tells you the volume of that generator. Zero (>0) is the *maximum volume* and >F the *minimum volume,* which turns off that generator.

Suppose the computer is playing a tone on generator 2. The following DATA will change its volume:

```
DATA >01B6,>05 ... and so on
```

>01 specifies that only one byte (the volume byte) must be read into the sound processor. >B6 shows the new volume (6) of generator 2 (B). >05 is the duration, and the ellipsis indicates that the sound list continues.

Once all the sounds have played, you have to stop them. All you have to do is specify the minimum volume for each of the generators you want to turn off, and a duration of 0. To turn off generator 3 (>D), you would use:

```
DATA >01DF,>00 ...
```

The DATA line shows that only one byte will be read, and that generator 3's volume is set to minimum.

To turn off all four generators, minimum volume (>F) is specified for each one. It would take four bytes and would look like this:

```
DATA >049F,>BFDF,>FF00
```

If you don't turn off the sound this way, the last sound will continue.

Loading the Sound DATA from BASIC

Sound tables can be loaded into VDP memory from BASIC with the CALL POKEV subroutine. That saves CPU memory for your own work. Sound lists can be loaded at any free VDP RAM address, providing they don't interfere with other data in memory. If your program works entirely in assembly language, the sound list can be loaded starting at VDP RAM address >1000.

Assume you wanted to load the data for one tone into VDP RAM at address 800 and the DATA was the following:

```
DATA >038D,>2592,>7800
```

First you need to translate each hexadecimal byte into its decimal value:

```
>03  = 3
>8D  = 141
>25  = 37
>92  = 146
>78  = 120
```

and then use CALL POKEV to POKE the decimal values into VDP RAM starting at location 800

```
CALL POKEV(800,3,141,37,146,120)
```

If your program runs entirely in assembly language, you can load the data table into VDP RAM by first writing it in CPU RAM and then using the VMBW utility to write it to VDP RAM.

Supposing the sound list is at CPU RAM address labeled SD and that it will be written to VDP address >1000:

```
    LI R0,>1000
    LI R1,SD
    LI R2,n           (Where n is the number of bytes in the
                      sound list)
    BLWP @>6028
      .
      .
      .
SD  DATA              (Sound data starts here)
```

Assembly Language Sound Routine

When the data table for the sound has been loaded into memory, you're ready to create the sound.

First of all, we must place the address in VDP RAM of the sound data at CPU address >83CC. If the data started at location >1000, you could use:

```
LI R9,>1000
MOV R9,@>83CC
```

Once >83CC is loaded with the data table location, place a value of >01 in address >83CE for the sound generation to begin. Finally, the right-hand bit of byte >83FD must be set to indicate that the sound list is in VDP RAM. To set this bit without disturbing the others in the byte, use the instruction SOCB (Set Ones Corresponding, Byte), which sets those bits in the second byte which are also set in the first byte. SOCB does not disturb any bits already set in that second byte. To set bit 7 of >83FD, then, you can use SOCB, comparing it to a byte which has a value of >01. The right-hand bit in >83FD will be set and the remaining bits left unchanged. Here's how it would look in an assembly language program segment:

```
    SOCB @CT,@>83FD
      .
      .
      .
CT  DATA >0100
```

The sound is then started by enabling VDP interrupts with the LIMI 2 instruction. You must create a loop to wait for the sound list to be played entirely. When this happens, the value in >83CE is set to zero. You can then enter:

```
     CLR R7
LP   LIMI 2
     LIMI 0
     CB R7, @>83CE
     JNE LP
       .
       .
       .
     (Program continues, sound has been played)
```

Playing Three Simultaneous Tones

The following program plays three tones simultaneously for 4.25 seconds at maximum volume and then stops. The sound list was found by using the "Creating DATA" utility program from this chapter. To stop the sound, the program places the minimum volume in generators 1, 2, and 3 with a duration of zero. No tones need to be specified when the sound is turned off.

To specify minimum volume for generator 1, use byte >9F. For generator 2, byte >BF and for generator 3, byte >DF. (To make the noise generator silent, specify minimum volume with byte >FF.) The program thus loads three volume bytes with a duration byte of zero (>03,>9F,>BF,>DF,>00) after the entry for the simultaneous tones in the sound list.

Multiple Assembly Language Tones

7D00	LWIP >70B8	(Load memory area for registers)
7D04	LI R0,>1000	(Sound list will be written to VDP RAM >1000)
7D08	LI R1,SL	(Sound list is at SL in CPU RAM)
7D0C	LI R2,16	(Sound list is 16 bytes long)
7D10	BLWP @>6028	(Write data to VDP RAM)
7D14	MOV R0,@>83CC	(Load position of sound list into >83CC)
7D18	MOVB @CV,@>83CE	(Load >83CE with value of >01 to start sound generation)
7D1E	SOCB @CV,@>83FD	(Set bit 7 of byte >83FD by comparing it to the byte at CV which has only bit 7 set)
7D24	CLR R7	(Clear R7 for later comparison)

```
7D26   LP   LIMI 2              (Enable VDP interrupts)
7D2A        LIMI 0              (Disable VDP interrupts)
7D2E        CB R7,@>83CE        (Is >83CE zero—sound list
                                finished?)
7D32        JNE LP              (Not yet, stay in loop)
7D34        JMP $               (Sound finished. Stop program
                                with endless loop)
7D36   CV   DATA >0100          (Data to be used to prepare
                                sound generation)
7D38   SL   DATA >0989,>3F90,>A92F,>B0CB (Sound list to be
                                executed)
7D40        DATA >23D0,>FF03,>9FBF,>DF00
```

When you run the program, you'll hear three tones together for 4.25 seconds.

Generating a Noise

Noises are generated in the same way as tones. The program below is exactly the same as that above; only the sound list has to be changed.

This program generates a noise at frequency −3 for two seconds and then stops. For explanations of the instructions, see the program "Multiple Assembly Language Tones."

Noise

```
7D00        LWPI >70B8
7D04        LI R0,>1000
7D08        LI R1,SD
7D0C        LI R2,7             (The sound list has seven bytes)
7D10        BLWP @>6028
7D14        MOV R0,@>83CC
7D18        MOVB @CV,@>83CE
7D1E        SOCB @CV,@83FD
7D24        CLR R7
7D26   LP   LIMI 2
7D2A        LIMI 0
7D2E        CB R7,@>83CE
7D32        JNE LP
7D34        JMP $
7D36   CV   DATA >0100
```

```
7D38   SD  DATA >02E2,>F078
7D3C       DATA >01FF,>0000
```

The noise is stopped by specifying minimum volume of the noise generator with byte >FF, added at the end of the list (address 7D3C).

Three Continued Tones

You can even play several tones, one after the other. The changes at the end of this section make the previous program play three tones, one by one, and then stop. It's equivalent to the BASIC program:

```
100 CALL SOUND(500,131,0)
110 CALL SOUND(800,330,5)
120 CALL SOUND(1200,220,2)
```

Since single tones are played, only sound generator 1 is used. The data for each tone would be:

```
>0386,>3590,>1E00
>0383,>1592,>3000
>038C,>1F91,>4800
```

and the data to stop sound generator 1 is:

```
>019F,>0000
```

But the zero bytes (00) at the end of each DATA line must be excluded. So the lines *should* look like this:

```
>0386,>3590,>1E03
>8315,>9230,>038C (18 bytes of sound data)
>1F91,>4801,>9F00
```

As the tones are played one after the other, the tone bytes are loaded in groups into generator 1.

Use the last program, "Noise," to actually play the sounds. The only values you have to change are the number of bytes to write into VDP RAM and the sound list. The former are loaded into R2 in address >7D0C:

```
 .
 .
 .
7D0C        LI R2,18
 .
 .
 .
```

```
7D38   SD   DATA >0386,>3590,>1E03,>8315
7D40        DATA >9230,>038C,>1F91,>4801,>9F00
```

Simultaneous Sound and BASIC

On the TI, you can play a melody or other sounds and have your BASIC program working at the same time, since the BASIC interpreter continues to process information while sound is generated. If you start playing a sound list and then return to BASIC, your program will continue executing while the sound plays.

The only problem that can crop up when using this technique is that VDP interrupts must be enabled upon return to BASIC to allow sound. If your BASIC program starts operating on VDP RAM, strange things might happen. Really the only way to use this method is to just experiment, looking for best results.

Sound and BASIC

This program links a BASIC program and an assembly language program to start playing sound. Control then returns to BASIC, which continues normally. Sound stops when the list is finished or when you generate some other sound from BASIC to interrupt the reading of the list.

```
7D00      LWPI >70B8        (Load memory area for registers)
7D04      LIMI 0            (Make sure VDP interrupts are
                            disabled while accessing VDP
                            RAM)
7D08      LI R0,>1000       (Position in memory where the
                            sound list will be placed)
7D0C      LI R1,SL          (Sound list is located in CPU
                            RAM at label SL)
7D10      LI R2,158         (Number of bytes in the sound
                            list)
7D14      BLWP @>6028       (Write sound list to VDP RAM)
7D18      MOV R0,@>83CC     (Load position of the sound list in
                            >83CC)
7D1C      MOVB @CB,@>83CE (Start sound generation)
7D22      SOCB @CB,@>83FD (Sound list is in VDP RAM)

7D28      LIMI 2            (Enable interrupts for sound to be
                            generated)
```

```
7D30        B *R11              (Return to BASIC)
7D32   CB   DATA >0100          (Comparison byte for sound
                                initialization)
7D34   SL   DATA >038B,>2390,>1003,>8B1A,>9010 (Sound
                                list to be played)
7D3E        DATA >038B,>1A90,>1003,>8C17,>9010
7D48        DATA >038C,>1790,>1003,>8315,>900A
7D52        DATA >038D,>1190,>0A03,>8315,>900A
7D5C        DATA >038B,>1A90,>1003,>8B23,>9010
7D66        DATA >038B,>1A90,>1003,>8B1A,>9010
7D70        DATA >038C,>1790,>1003,>8C17,>9010
7D7A        DATA >0383,>1590,>2003,>8B1A,>9010
7D84        DATA >038B,>2390,>1003,>8B1A,>9010
7D8E        DATA >038B,>1A90,>1003,>8C17,>9010
7D98        DATA >038C,>1790,>1003,>8315,>900A
7DA2        DATA >038D,>1190,>0A03,>8315,>900A
7DAC        DATA >038B,>1A90,>0703,>8B1A,>901D
7DB6        DATA >038E,>0F90,>1E03,>8C17,>9010
7DC0        DATA >0381,>1490,>1003,>8315,>901E
7DCA        DATA >038B,>1A90,>1601,>9F00
7DD2        AORG >701E
701E        DATA >7FE0
7020        AORG >7FE0
7FE0        TEXT 'SOUND '       (Add name and position of the
                                program to the REF/DEF Table)
7FE6        DATA >7D00
7FE8        END
```

Once you've entered the program, press FCTN = (QUIT) and select TI BASIC. In direct mode, type:

CALL LINK("SOUND")

The melody "Pop Goes the Weasel" will play at the same time the BASIC interpreter is working. The cursor will appear in its home position as soon as sound begins. You can write a program while the melody is playing, list a program in memory, or whatever you wish.

Sound can be stopped with any CALL SOUND statement:

CALL SOUND(−1,−1,30)

You can also check to see if the program has finished playing the sound list by checking the value of the byte in address >83CE. Use CALL PEEK to do this; if the value returned is zero, the sound list is finished.

The value >83CE in decimal is 33742. Remember, though, that you have to subtract 65536 from any value greater than 32767 to arrive at the correct value to use in a CALL PEEK. 33742 minus 65536 equals −31794. That's the value to use.

```
CALL PEEK(-31794,X)
IF X=0 THEN ... (sound list is finished)
```

The following BASIC program plays the sound list loaded previously into memory over and over until a key is pressed:

```
100 CALL LINK("SOUND")
110 CALL KEY(0,K,S)
120 CALL PEEK(-31794,X)
130 IF X=0 THEN 100
140 IF S=0 THEN 110
150 CALL SOUND(-1,-1,30)
160 END
```

Endless Possibilities

The possibilities of using sound on your TI are almost endless. But sound has really only been introduced in this chapter. Direct access of the sound and noise generators gives you greater control on all sounds executed, and the fact that you can play complete melodies at the same time your program is running gives you even more flexibility. Don't be afraid to experiment when creating sounds—you'll learn and have fun at the same time.

Chapter 11

Graphics Modes on the TI

Graphic Modes on the TI

Not only can your TI-99/4A display sprites and play sound and music, it's also an excellent graphics machine. Assembly language programming gives you even faster access to these graphics capabilities, allowing you to do things quickly and easily.

The TI has four different graphics modes: *text, graphics, multicolor,* and *bitmap.* In this chapter, you'll learn about each mode's features, as well as see examples of each. Most importantly, you'll see how the VDP write-only registers are used.

The VDP Write-Only Registers

The Video Display Processor (VDP) chip in your computer uses a set of 8 registers, different from the 16 registers you've been using up to now. These 8 registers are called the *VDP write-only registers.* They contain information about screen color, current graphics mode, sprite magnification, and table positions, among other things. Let's take a closer look at the registers.

Changing Values

Each VDP register holds one byte of information. This byte contains a lot of information, for each of its eight bits—depending on whether each is set to a 1 or 0—can hold a different piece of data. To change the value of a VDP register, you must first determine which bits of the byte must be *set* (contain a 1) and which *reset* (contain a 0).

Assume you want VDP register 3 to have bits 3, 5, and 6 set and the rest reset. The byte would look like this:

Bits:	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
Bit Value:	**128**	**64**	**32**	**16**	**8**	**4**	**2**	**1**
	0	0	0	1	0	1	1	0

Thus the binary number to write into VDP register 3 would be:

00010110

To arrive at the decimal value of the byte, simply add the values of all bits that are set.

0 + 0 + 0 + 16 + 0 + 4 + 2 + 0 = 22 (decimal)

Decimal 22 is the same as >16 (hexadecimal). To write the new byte value (>16) into VDP register 3, use the VWTR (VDP Write To Register) utility at address >6034. Your workspace register 0 (R0) must have its left byte loaded with the VDP register you want to change (>00–>07, to indicate the eight VDP registers), and the right byte loaded with the new value you want to assign (in this case, >16). So you'd enter:

LI R0,>0316	(Load R0 with VDP register [>03] and the value to write to it [>16])
BLWP @>6034	(Write the new value to VDP register 3)

VDP Registers Descriptions

VDP register 0. Bits 0–5 of this register are not used and must be reset (a 0 in each). Bit 6 determines whether the computer is in bitmap mode. If the bit is set, the computer is in bitmap mode. Bit 7 enables external video when set and disables it when reset. This bit is reset (0) for most applications.

VDP register 1. It contains memory, screen, and graphics mode information. When you change the value in this register, the new value must also be placed at CPU RAM address >83D4. If you don't, the register is reset to its original value when a key is pressed. The computer writes the value at location >83D4 into VDP register 1 each time a key is pressed. That's how the register is constantly updated. This process is necessary only for this VDP register.

The eight bits of VDP register 1 tell you the following:

If bit 0 is set, it indicates you're executing a 16K RAM operation; if it is reset, a 4K RAM operation. Leave it set for your applications.

If bit 1 is set, display on the screen is visible. If reset, anything on the screen is transparent and all you'll see is the screen color. Set this bit in your applications.

Bit 2 allows VDP interrupts if it's set and does not if it's reset.

Bit 3 puts the computer in text mode if the bit is set.

Bit 4 places the computer in multicolor mode if it's set.

Bit 5 is not used and you should leave it reset.

Bits 6 and 7 contain sprite information. If bit 6 is reset, you're using normal-sized sprites; if it's set, enlarged (four characters in area) sprites. If bit 7 is reset, you're using unmagnified sprites; if set, magnified sprites. This is explained in more detail in Chapter 9.

VDP Register 1

Bit Number	Effect When Set	Effect When Reset
0	16K RAM	4K RAM
1	Screen display visible	Screen display transparent
2	Enable VDP interrupts	Disable VDP interrupts
3	Text mode	
4	Multicolor mode	
5	(Should be left reset)	
6	Enlarged sprites	Normal-sized sprites
7	Magnified sprites	Unmagnified sprites

VDP register 2. This register indicates the location of the table describing the screen. As you've seen in previous chapters, to display something on the screen, you print it in any of 768 positions, 0 through 767. What you were actually doing was writing that character to the Screen Image Table, which starts at VDP address >00. Because this table represents the screen, whatever you write to it is visible.

This table does not necessarily need to occupy the first positions in VDP RAM and can be located in other areas, according to the value which is found in this register. The default value of >00 informs the computer that the table is located starting at location >00.

The table's starting address can be found by multiplying the value in this register by >400 (1024 decimal). A value of >00 here makes the table begin at >0000 (>00 × >0400), while a value of >01 starts the table at >0400 (>01 × >0400). For more information, see the section "Moving Tables."

VDP register 3. This register contains the starting location of the Color Table. To calculate the table's starting address, the value in this register is multiplied by >40. A value of >02 starts the Color Table at >80 (>02 x >40).

VDP register 4. This register holds the starting address of the Pattern Descriptor Table. Multiply the value here by >0800 to find the starting location. If >00 is here, the table begins at >0000. A >01 indicates >0800 is the first address, a >02 means the table begins at >1000, and so on.

VDP register 5. Register 5 lets you move the Sprite Attribute List in memory. To find where the table begins, multiply the value in the register by >80. For the Sprite Attribute List to start at >0100, for instance, a >02 would be loaded into this register.

VDP register 6. It defines the starting address of the Sprite Descriptor Table in VDP memory. The starting address is the product of the value in the register and >800. For example, >02 in this register places the Sprite Descriptor Table from address >1000 on.

(Note: The Sprite Motion Table cannot be moved in memory, and always has a starting address of >0780.)

VDP register 7. This register contains the screen color information. If the computer is set in text mode, the left digit of the byte defines the color of all characters on the screen. In any of the graphics modes, the right digit of the byte indicates the background color, or the color of the screen. A value of >A1, for instance, creates yellow characters on a black screen in text mode, or makes the screen blank in any of the other modes.

Moving Tables

Different information tables in VDP RAM can be moved around simply by changing the value in a VDP register. For many applications, moving the tables will be necessary, but you should try to avoid doing so if possible. It just complicates things.

A table you *can* safely move is the Screen Image Table, which represents the screen. But why would you want to move this particular table? Let's look at a good reason.

If you wish to instantly change from one screen to another, such as from one detailed graphics picture to the next, you can draw the second screen in another memory area and then just change the value in VDP register 2 to point to the second picture. The first screen will be instantly replaced by the second.

If you've never seen this done, the best way to explain it is to show you an example. The next program displays THIS IS SCREEN ONE on the visible screen (0–767), and places a second message in a free memory area starting at >1000. The second screen will not be visible. Once a key is pressed, the pointer to the screen table in VDP register 2 changes to point to the memory area which contains the second message. The first message is instantly replaced by the second.

The original message will still be in memory, just not visible on the screen. Though you're only changing a screen message here, keep in mind that complete graphics scenes can be placed in memory and the entire display changed by altering the pointer value in VDP register 2. The process would be just the same.

Screen Switching

```
7D00        LWPI >70B8        (Load the memory area for the
                              registers)
7D04        LI R0,>012C       (Screen position [300] to display
                              text)
7D08        LI R1,T1          (Text at CPU address labeled T1)
7D0C        LI R2,18          (Length of the text)
7D10        BLWP @>6028       (Write multiple bytes to VDP
                              RAM)
7D14        LI R0,>112C       (Position in VDP RAM where the
                              second message will be written)
7D18        LI R1,T2          (Position of the second text in
                              CPU RAM)
7D1C        LI R2,9           (Length of the second text)
7D20        BLWP @>6028       (Write the second text to VDP
                              RAM. It won't be visible because
                              the Screen Image Table starts at
                              >00 and goes to >0300)
7D24        CLR @>8374        (Standard keyboard scan)
7D28        CLR R1            (Prepare R1 to receive the status
                              byte in the KSCAN loop)
7D2A   LP   BLWP @>6020       (Branch to execute the KSCAN
                              routine)
7D2E        MOVB @>837C,R1    (Move the status byte into the left
                              byte of R1)
7D32        COC @BT,R1        (Compare Ones Corresponding
                              with the value at BT)
7D36        JNE LP            (No key has been pressed. Return
                              to KSCAN loop)
```

```
7D38        LI R0,>0204       (Key has been pressed. Prepare to
                              write >04 into VDP R2 because
                              >04 X >0400 = >1000. The
                              Screen Image Table will then be-
                              gin at >1000)
7D3C        BLWP @>6034       (Write the value to the VDP
                              register)
7D40        JMP $             (Stop program with an endless
                              loop)
7D42   BT   DATA >2000        (Comparison value for the
                              KSCAN loop)
7D44   T1   TEXT 'THIS IS SCREEN ONE' (First text to be
                              displayed)
7D56   T2   TEXT 'SCREEN #2' (Second message to be
                              displayed)
```

When you run the program, the message THIS IS SCREEN ONE will appear on the screen. You can see the message because the Screen Image Table is located from >00 to >0300 and the message prints from position >012C. The second message (SCREEN #2) is also placed in VDP RAM, but at >112C, an area *not* on the screen. When you press any key, the pointer to the Screen Image Table is changed to display the bytes from >1000 to >1300. This makes the first message vanish (though it's still in memory, where you wrote it initially) and replaces it with the second message, now in a visible memory area. You'll also see the words *Line-by-Line Assembler* and © *1982 TI,* unless you've overwritten them with something else.

Again, try to avoid moving the other tables unless they have to be located at another area and the documentation tells you where. If you don't need to move the tables, but you want to relocate them, it's a good idea to experiment until you're sure where the tables are, and you're certain they won't interfere with each other, or with other data in VDP memory.

Graphics Mode

When the computer is in graphics mode, the screen is divided into 32 columns by 24 lines. You can define and set the colors of each of 255 characters, and you can use sprites. The graphics mode is the one used by BASIC and by most applications.

Using the graphics mode has already been described in Chapter 8.

Text Mode

When the computer is in text mode, the screen is divided into 40 columns by 24 lines. To set the computer in text mode, bit 3 of VDP register 1 must be set (contain a 1). The Pattern Descriptor Table is the same as in graphics mode, but each character has a 6 × 8 pixel definition. The Screen Image Table also changes; it doesn't run from 0 to 767 as in graphics mode, but from 0 to 959. Text mode has 960 screen positions.

You can't use sprites in text mode and you have only two available colors: foreground and background. All characters are the same color.

The color used in text mode is set in VDP register 7. The left digit of the byte is the foreground color and the right digit the background color. An example of a program working in text mode is the *Line-by-Line Assembler.*

Remember that when you change the value of VDP register 1, to set the computer in text mode or for some other reason, you should copy the byte and place the value at CPU address >83D4.

Setting Text Mode

The next program sets the computer in text mode, makes the foreground color black (all text will be black) and the background color light red (screen color). The computer automatically sets all predefined characters in a 6 × 8 pixel matrix for use in text mode. Any key you press prints its character on the screen, starting at screen position 0.

To set the computer in text mode, bit 3 of VDP register 1 must be set. Bits 0, 1, and 2 are set (they are left set for most applications). Bits 4, 5, 6, and 7 should be reset. The value to be written in VDP register 1 would then be:

11110000 = 128 + 64 + 32 + 16 = 240 = >F0

7D00	**LWPI >70B8**	(Registers store values starting at >70B8)
7D04	**LI R0,>0719**	(Prepare to set the screen and text color. Write the color black [>1] on light red [>9] into VDP register 7)

```
7D08      BLWP @>6034       (Write the color byte to the VDP
                            register)
7D0C      LI R0,>01F0       (Prepare to write >F0 to VDP reg-
                            ister 1 to set text mode)
7D10      BLWP @>6034       (Write the value to the register)
7D14      SWPB R0           (Place the new value written to
                            VDP register 1 in the left byte of
                            R0)
7D16      MOVB R0,@>83D4    (Place it in >83D4 so the com-
                            puter will reset the value of VDP
                            register 1 each time a key is
                            pressed)
7D1A      CLR @>8374        (Standard keyboard scan)
7D1E      CLR R0            (First character to be printed in
                            the first screen position)
7D20      CLR R1            (Prepare R1 to receive the status
                            byte)
7D22  LP  BLWP @>6020       (Branch to scan the keyboard)
7D26      MOVB @>837C,R1    (Move the status byte to R1)
7D2A      COC @BT,R1        (Compare One Corresponding of
                            the comparison value of BT and
                            the left byte of R1)
7D2E      JNE LP            (If no key has been pressed, stay
                            in the KSCAN loop)
7D30      MOVB @>8375,R1    (Move the ASCII code of the key
                            pressed to the left byte of R1)
7D34      BLWP @>6024       (Write the byte on the screen)
7D38      INC R0            (Increment the screen printing
                            position)
7D3A      CI R0,961         (Has the last screen position been
                            passed?)
7D3E      JLT LP            (If not, return to the KSCAN loop)
7D40      CLR R0            (If it has, reset printing position to
                            the top-left corner)
7D42      JMP LP            (Return to the keyboard scanning
                            loop)
7D44  BT  DATA >2000        (Comparison value for the key-
                            board scan)
```

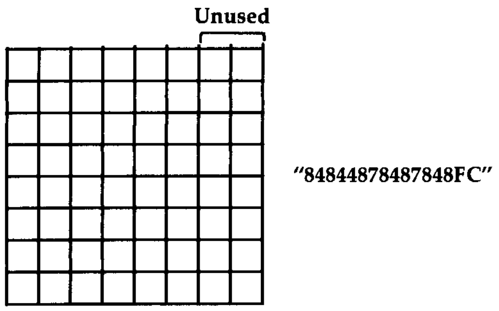
When you end and run this program, the screen turns light red and all text on the screen appears in black. Whatever key you press will display its character on the screen.

If you execute the program with EASY BUG's E command, as soon as you press ENTER, the ?E7D00 message jumps up and to the right. That's because the graphics mode screen position of the message is different from the corresponding position in the text screen. Text mode, remember, has 192 more character positions on its screen than the graphics mode.

Defining Characters in Text Mode

Characters are defined in the same way in text mode as they are in the graphics mode. Definitions are written into the Character Table (see Chapter 8). The only difference is that characters in text mode are defined in a 6 × 8 pixel grid, two pixels narrower than the grid used in graphics mode. This is really no problem; just define the character in an 8 × 8 pixel grid without using the two right-hand columns. When writing the character definition, imagine these pixels as being "off," as illustrated in the figure below.

6 × 8 Pixels



Redefined Asterisk

This program defines character 42 (the asterisk) as a box and then displays it on the screen.

7D00	LWPI >70B8	(Memory area registers will use)
7D04	LI R0,>071F	(Prepare colors for the text mode)
7D08	BLWP @>6034	(Write the color byte to VDP register 7)
7D0C	LI R0,>01F0	(Prepare to set the computer in text mode)

```
7D10        BLWP @>6034        (Write the value to VDP register
                               1)
7D14        SWPB R0            (Prepare to write the value to
                               >83D4)
7D16        MOVB R0,@>83D4     (Place the left byte of R0 in
                               >83D4)
7D1A        LI R0,>0950        (Load the position of character 42
                               in the Pattern Descriptor Table)
7D1E        LI R1,DF           (Load the position of the defi-
                               nition in CPU RAM)
7D22        LI R2,8            (Definition is eight bytes long)
7D26        BLWP @>6028        (Write the definition to the table)
7D2A        LI R0,170          (Load character's print position)
7D2E        LI R1,>2A00        (Load the character code)
7D32        BLWP @>6024        (Print the character on the screen)
7D36        JMP $              (Stop program execution)
7D38   DF   DATA >FC84,>8484,>8484, >84FC (Character
                               definition)
```

Multicolor Mode

If your computer is set in multicolor mode, the screen displays small, colored boxes. You *can* set the color of each of these boxes. The screen itself is divided into 64 columns and 48 rows. Each of the 3072 positions is a 4 × 4 pixel box, assigned with a color. Sprites may be used. The colors of every two boxes are described by one byte; since the screen has 3072 boxes, you need 1536 bytes to describe the screen colors. The Pattern Descriptor Table stores the bytes which define the colors. The left digit of each byte holds the color of one box, and the right digit the color of the box immediately to the right of the first box.

So that each screen position can be identified, you have to number the screen positions from 0 to 31 for the first four rows. That takes care of one 128-byte segment. The next four screen rows are numbered from 32 to 61. That's another 128-byte segment. This numbering scheme continues to the last 128-byte segment, the last four screen rows, which hold numbers 160 to 191. The following shows how the screen is numbered:

Rows	Screen Position Numbers		Number of Bytes
1–4	0 1 2 3	31	128
5–8	32 33 34 35	63	128
9–12	64	95	128
13–16	96	127	128
17–20	128	159	128
21–24	160	191	128

This program segment numbers the multicolor screen in the above manner. Only the instructions and their explanations are listed, not addresses for those instructions.

	CLR R0	(Start at screen position 0)
	LI R7,6	(Screen is divided into six 128-byte segments)
	CLR R5	(Initial value of each 128-byte segment [0, 32, 64, 96, 128, and 160] will be kept in this register)
L1	**LI R3,4**	(Each 128-byte segment is divided into four 32-byte groups)
L2	**LI R4,32**	(Will write 32 bytes on each line)
	MOVB R5,R1	(Move the value to be written on that screen position into R1 for the VSBW routine)
L3	**BLWP @>6024**	(Write the value to the Screen Image Table)
	INC R0	(Increment screen printing position)
	AI R1,>0100	(Add one to the value to be written on the screen)
	DEC R4	(Decrease number of bytes left to write on that line; stay in loop L3)
	JNE L3	(If there are still bytes left to write on that line, stay in loop L3)
	DEC R3	(Decrease number of 32-byte groups in the 128-byte segment left to write)
	JNE L2	(If the four 32-byte groups have still not been written, stay in loop L2)
	AI R5,>2000	(Finished with the 128-byte segment. The numbering of the next segment will start with a value 32 greater than the previous segment)

```
     DEC R7             (Decrease number of 128-byte segments
                        left to write)
     JNE L1             (If there are segments left, stay in loop
                        L1)
```

To clear the screen when you select multicolor mode, you must make all the squares transparent. As the Pattern Descriptor Table, which now stores colors, starts at >0800 and is 1536 bytes long (extends up to >0E00), the following instructions write the value of >00 into each of the 1536 bytes of the table:

```
     LI R0,>0800        (Table starts at >0800)
     CLR R1             (Value to be written to the table is 0)
LP   BLWP @>6024        (Write >00 to VDP RAM)
     INC R0             (Increase the printing position in the
                        table)
     CI R0,>0E00        (Has the end of the table been
                        reached?)
     JNE LP             (If not, stay in the loop)
```

To set the computer in multicolor mode, set bit 4 of VDP register 1 by writing the value >E8 into it, as well as saving a copy of that value in address >83D4. These instructions do that:

```
     LI R0,>01E8        (Prepare to write >E8 into VDP register
                        1)
     BLWP @>6034        (Write the value to the VDP register)
     SWPB R0            (Move the value to the left byte of R0)
     MOVB R0,@>83D4     (Write it to location >83D4)
```

An Immediate Example

You can easily see an example of multicolor mode. Since this mode is set by writing >E8 to VDP register 1, and each time a key is pressed the computer places a copy of the value at >83D4 into that register, you can load the value of >E8 into address >83D4 from BASIC, then press a key. The computer will be in multicolor mode. Try this: With the *Mini Memory* cartridge in place, select TI BASIC. Type in direct mode:

```
CALL LOAD(-31788,232)
```

(You loaded the value into −31788 because >83D4 is 33748 in decimal. Since 33748 is greater than 32767, you need to subtract 65536 from it to arrive at the correct number.)

The value to select multicolor mode is >E8 (232 in decimal). When you've typed in the CALL LOAD statement, press ENTER and then hit any key. Each character on the screen will be instantly transformed into four small colored squares. Whatever you type on the screen will appear in these colored squares. To return the computer to normal, write the value of >E0 (to reset bit 4) into >83D4. The computer will write that value to VDP register 1 when a key is pressed. >E0 is 224 in decimal, so blindly type (you'll see only colored squares, remember):

CALL LOAD(−31788,224)

and press ENTER. When you hit another key, the computer will be reset to graphics mode.

Finding the Correct Byte in the Table

Now that you've established a numbering system for the multicolor screen, you have to be able to determine a particular byte's position on that screen. The calculations aren't that difficult. Here's how to find the location of the byte in the Pattern Descriptor Table which defines the color of a box.

Assume variable X contains the column (0–63) and Y the row (0–47) of the square whose color you want to change. The calculations to find the position in the table which corresponds to that byte are:

```
X1 = X/2
XF = INT(X/2)
R1 = X1−XF

Y1 = Y/8
YF = INT(Y/8)
R2 = Y1−YF
Q = >0800+YF*256+XF*8+R2  (>0800 is the position where the
                           Pattern Descriptor Table begins
                           when the program has not been
                           called from BASIC)
```

The value Q is the position in the table of the byte you want to change. Once you've found the byte, you have to determine which *digit* of that byte represents the box you want to alter. R1's value indicates this. If the value in R1 is 0, change the *left* digit. If it's 1, change the *right* digit.

Let's see an application of this procedure. "Box Draw," the program below, runs entirely in assembly language and lets you draw colored lines, four pixels wide, on the screen. The arrow keys control the line's direction. In effect, it's an electronic doodle pad. Pressing the 1 key changes the screen color, pressing 2 alters the color of the lines (from that moment on), and pressing the 3 key clears the screen. Pressing "FCTN = (QUIT) returns you to the master screen.

Box Draw

The program listing has been divided into segments, each part generally described, with a short explanation included beside each instruction. Note that the computer is set into text mode at the beginning of the program, and is left that way until the screen initialization is finished. This is done so the user won't see all the screen numbering and clearing processes taking place.

The first section directs the computer to the correct place in memory and sets the computer in text mode, with background and foreground colors both black. The program uses 16 labels, so it starts at address >7D20 to leave room for 17. Remember that the last is a null entry (see Chapter 4, the section named "Saving Memory: Fewer Labels," for more details).

7D00	**AORG >7D20**	(Prepare to start program at address >7D20)
7D20	**LWPI >70B8**	(Load memory area for the registers)
7D24	**CLR @>8374**	(Standard keyboard scan)
7D28	**LI R0,>0711**	(Prepare to make the screen black on black)
7D2C	**BLWP @>6034**	(Write the color byte to VDP register 7)
7D30	**LI R0,>01F0**	(Prepare to write >F0 to VDP register 1)
7D34	**BLWP @>6034**	(Set the computer in text mode)

Initialize the screen by numbering it in 128-byte segments as earlier explained. The next three loops do this.

7D38	**CLR R0**	(Start at screen position 0)
7D3A	**LI R7,6**	(Six 128-byte segments to write)
7D3E	**CLR R5**	(R5 will control the value to be written to the screen)

```
7D40  L1  LI R3,4          (Four lines in each 128-byte
                           segment)
7D44  L2  LI R4,32         (32 characters on each line)
7D48      MOVB R5,R1       (More values to be written to the
                           screen for the VSBW routine)
7D4A  L3  BLWP @>6024      (Writes the value to the screen)
7D4E      INC R0           (Increase by one the value to be
                           written to the screen)
7D54      DEC R4           (Decrease number of bytes
                           remaining to be written on that
                           line)
7D56      JNE L3           (If the end of the line has not
                           been reached, stay in loop L3)
LD58      DEC R3           (Decrease number of lines remain-
                           ing in the 128-byte segment)
7D5A      JNE L2           (If there are still lines left in the
                           segment, stay in loop L2)
7D5C      AI R5,>2000      (The numbering of the next seg-
                           ment will start at a value 32
                           greater than the previous one)
7D60      DEC R7           (Decrease number of 128-byte
                           segments left)
7D62      JNE L1           (If there are still segments left,
                           stay in loop L1)
```

The next step is to clear the Pattern Descriptor Table, where the colors of the boxes on the screen are kept. The program will first make all the boxes transparent.

```
7D64  CL  LI R0,>0800      (The Pattern Descriptor Table
                           starts at >0800)
7D68      CLR R1           (Color to be written is >00
                           [transparent])
7D6A  LP  BLWP @>6024      (Write the color to the table)
7D6E      INC R0           (Increment position in the table)
7D70      CI R0,>0E00      (Has the end of the table been
                           reached?)
7D74      JNE LP           (If not, stay in the clearing loop)
```

Before the main execution loop begins, multicolor mode is set by writing >E8 to VDP register 1 (the value is also stored at CPU address >83D4). The initial column of the first block is set in R3, the initial row of the first block in R4, the initial screen color in R5, and the initial block color in R14.

```
7D76        LI R0,>01E8         (Prepare to write >E8 to VDP reg-
                                ister 1)
7D7A        BLWP @>6034         (VDP Write to Register to set the
                                computer in multicolor mode)
7D7E        SWPB R0             (Prepare to write >E8 to address
                                >83D4)
7D80        MOVB R0,@>83D4      (Move the left byte of R0 to
                                >83D4)
7D84        LI R3,32            (Column of the initial square)
7D88        LI R4,24            (Row of the initial square)
7D8C        LI R5,>0001         (Black is initial screen color)
7D90        LI R14,>9000        (Initial block color)
```

Multicolor is set. The following section scans the keyboard and branches to the corresponding routine, depending on the key pressed.

```
7D94   LL   LIMI 2              (Enable interrupts so the program
                                can be stopped with FCTN =
                                (QUIT))
7D98        LIMI 0              (Disable VDP interrupts again)
7D9C        LI R13,2000         (Delay loop)
7DA0        DEC R13             (Decrease value in R13)
7DA2        JNE $-2             (If not zero, delay loop not
                                finished)
7DA4        BLWP @>6020         (Scan the keyboard)
7DA8        CLR R1              (Prepare R1 to receive the ASCII
                                of the key pressed)
7DAA        MOV @>8375,R1       (Move the ASCII code of the key
                                pressed into the right byte of R1)
7DAE        CI R1,83            (Has the S [left arrow] been
                                pressed?)
7DB2        JEQ LT              (If so, jump to label LT)
7DB4        CI R1,68            (Has the D [right arrow] been
                                pressed?)
7DB8        JEQ RT              (If it has, jump to label RT)
7DBA        CI R1,69            (Has the E [up arrow] been
                                pressed?)
7DBE        JEQ UP              (If it has, jump to label UP)
```

```
7DC0         CI R1,88       (Has the X [down arrow] been
                            pressed?)
7DC4         JEQ DN         (If so, jump to label DN)
7DC6         CI R1,49       (Has the 1 [change screen color]
                            been pressed?)
7DCA         JEQ SC         (If it has, jump to label SC)
7DCC         CI R1,50       (Has the 2 [change block color]
                            been pressed?)
7DD0         JEQ BC         (If it has, jump to label BC)
7DD2         CI R1,51       (Has the 3 [clear screen] been
                            pressed?)
7DD6         JEQ CL         (If so, jump to label CL)
7DD8         JMP LL         (For any other key pressed, or no
                            key pressed at all, stay in the
                            KSCAN loop at LL)
```

Next enter the routines to increase or decrease the coordinates of the block which changes the color as various arrow keys are pressed.

```
7DDA  LT     DEC R3         (Left arrow pressed. Decrease col-
                            umn of block)
7DDC         CI R3,−1       (Has it passed column 0?—block
                            out of screen)
7DE0         JNE DR         (If not, jump to color the block at
                            label DR)
7DE2         CLR R3         (Block out of screen. Reset its
                            position)
7DE4         JMP DR         (Jump to color the block at DR)
7DE6  RT     INC R3         (Right arrow pressed. Increase col-
                            umn of the block)
7DE8         CI R3,64       (Has it passed the last onscreen
                            column [63]?)
7DEC         JLT DR         (If not, jump to color the block at
                            DR)
7DEE         LI R3,63       (Block out of screen. Reset its
                            position)
7DF2         JMP DR         (Jump to color the block at DR)
7DF4  UP     DEC R4         (Up-arrow key pressed. Decrease
                            block's row value)
7DF6         CI R4,−1       (Is it past the top row, row 0?)
```

```
7DFA         JNE DR            (If not, jump to color the block at
                               DR)
7DFC         CLR R4            (If it is, reset the row value of the
                               block)
7DFE         JMP DR            (Jump to color the block at DR)
7E00   DN    INC R4            (Down-arrow key pressed. In-
                               crease block's row value)
7E02         CI R4,48          (Check to see whether it's within
                               the screen bottom)
7E06         JLT DR            (If it is, jump to color the block at
                               DR)
7E08         LI R4,47          (If it's out of bounds, reset block's
                               row value)
7E0C         JMP DR            (Position reset. Jump to color the
                               block at DR)
```

The following routine changes the color of the screen each time the 1 key is pressed. In this routine, if the value of the color code for the screen is >F, it's reset to >0 before incrementing by one. The right byte of R5 will then be updated to the new color. This value is moved to R0 for the VWTR routine. >0700 is added to R0, for this writes the value >07 to the left byte of the register without disturbing the right byte. The VWTR routine is executed; the updated color byte will have been written to VDP register 7; and the screen color changed. A delay loop insures that the screen doesn't change colors too quickly.

```
7E0E   SC    CI R5,>000F       (Last color, white [>0F]?)
7E12         JNE $+4           (If not, jump to the updating
                               instruction)
7E14         CLR R5            (Update the color of the screen to
                               >00)
7E16         INC R5            (Increment R5)
7E18         MOV R5,R0         (Move R5 to R0 for the VWTR
                               routine)
7E1A         AI R0,>0700       (Write >07 to the left byte of R0
                               so the color byte [right byte] of R0
                               will be written to VDP register 7)
7E1E         BLWP @>6034       (Write the new screen color to
                               VDP register 7)
7E22         LI R13,20000      (Prepare for the delay loop)
7E26         DEC R13           (Decrease the value in R13)
```

```
7E28        JNE $-2          (If not zero, stay in the delay
                             loop)
7E2A        JMP LL           (Screen color changed. Return to
                             the main KSCAN loop)
```

Add the routine to change the block color each time the 2 key is pressed. Update the value of the block color controlled in R14. The routine at label DR changes the color of the block automatically. A delay loop is also added after each change so that it can be more easily seen.

```
7E2C   BC   AI R14,>1000     (Add 1 to the current color code.
                             If the old code is >F, it's auto-
                             matically reset to >0)
7E30        LI R13,20000     (Prepare for the delay loop)
7E34        DEC R13          (Decrease the value in R13)
7E36        JNE $-2          (If not zero, delay not finished, so
                             stay in loop. Otherwise, continue
                             execution with the DR routine to
                             change block color)
```

When the block has to be colored, the program calculates the position of the byte in the Pattern Descriptor Table corresponding to the block. (The following calculations were presented in the previous section "Finding the Correct Byte in the Table.")

```
7E38   DR   LI R15,2         (Load 2 into R15. R3 has to be di-
                             vided by two)
7E3C        MOV R3,R7        (Move the block's column to R7
                             for the division)
7E3E        CLR R6           (Prepare R6 for the division. R6
                             and R7 [000000xx, where xx is
                             the column value] will be divided
                             by the value in R15)
7E40        DIV R15,R6       (Execute the division. The quo-
                             tient is placed in R6 and the
                             remainder in R7)
7E42        MOV R4,R9        (Prepare to divide the row value
                             by eight. Place the row value in
                             R9)
7E44        CLR R8           (Clear R8 for the division)
7E46        LI R15,8         (Row of block will be divided by
                             eight)
```

7E4A	**DIV R15,R8**	(Execute the division. The quotient will be placed in R8 and the remainder in R9)
7E4C	**SLA R6,3**	(Multiply the quotient of the first division by eight by shifting every bit in R6 three positions to the left. See Chapter 6)
7E4E	**SLA R8,8**	(Multiply the quotient of the second division by 256 in the same way)
7E50	**A R6,R8**	(Add the result of both multiplications)
7E52	**A R8,R9**	(Add the remainder of the second division. The answer will be stored in R9)
7E54	**AI R9,>0800**	(Add the position in VDP memory where the pattern table begins. The position of the byte that controls the color of the block is stored in R9)
7E58	**MOV R9,R0**	(Move the value to R0 for the VSBR routine)
7E5A	**CLR R1**	(Value read from the table will be placed in R1)
7E5C	**BLWP @>602C**	(Read the color byte from the table)
7E60	**MOV R1,R10**	(Store the color byte in R10)

Now that the program's read the byte from the Pattern Descriptor Table, it must decide whether to change the right digit or the left digit of the byte. The program will consider only the left byte of all memory words. The right byte will be zero.

First of all, the unused digit, the digit to change, is cleared using the ANDI instruction. (See Chapter 12 for details of this instruction.) The new color digit is then written into the unoccupied location.

For example, suppose the color byte to change is >5300 (>53) and you want to exchange the 5 with a 1. First of all, the 5 is changed to a 0, so the color byte will be >0300. This is done with the logical instruction ANDI, comparing the color byte to be changed with the byte >0F00. The bits which are set in both bytes will be set in the new byte. All other bits will be reset. It works like this:

```
>5300: 0 1 0 1 0 0 1 1 0 0 0 0 0 0 0 0
>0F00: 0 0 0 0 1 1 1 1 0 0 0 0 0 0 0 0
```

results in

```
ANDI: 0 0 0 0 0 0 1 1 0 0 0 0 0 0 0 0
```

which is >0300.

In this way, the unnecessary digit of the color byte is cleared. Now the value 1 is added to complete the process:

>1000 + >0F00 = >1F00 (updated color byte)

When the other digit of the color byte has to be changed, the ANDI instruction is used with the old color byte and the byte >F000. For example, if the color byte is >1300 and the 3 has to be changed to 2:

$$\text{Bytes} \left.\begin{array}{r}\text{>1300}\\ \text{>F000}\end{array}\right\} \overset{\rightarrow}{\text{ANDI}} \quad \text{>1000}$$

$$\left.\begin{array}{r}\text{>1000}\\ \text{>0200}\end{array}\right\} \overset{\rightarrow}{\text{AB}} \quad \text{>1200}$$

In this way, you've been able to change one of two digits of a byte without disturbing the other. (If the ANDI instruction is still not clear, refer to Chapter 12 for a more detailed explanation.)

7E62	**CI R7,0**	(Check the remainder of the first division to determine which digit, left or right, has to be changed)
7E66	**JEQ LD**	(If the remainder is zero, the left digit has to be changed. Jump to label LD to do so)
7E68	**ANDI R10,>F000**	(Right digit has to be changed. Clear the right digit, but store the left digit unchanged by using the ANDI instruction)
7E6C	**SRL R14,4**	(The new color has to be written in the second digit of R10. At present it's stored in the first digit, so shift all binary digits four positions right, thus moving the color digit to the second position of R14)

```
7E6E        AB R14,R10         (Write the new color byte into the
                               right digit of the color byte in
                               R10)
7E70        SLA R14,4          (Move the color digit back to its
                               usual position, the first digit of
                               R14)
7E72   AL   MOV R10,R1         (Move the new color byte into R1
                               to write it to the Pattern Descrip-
                               tor Table)
7E74        BLWP @>6024        (Write it to the table. R0 is still
                               loaded with the correct address of
                               the byte)
7E78        JMP LL             (Jump back to the KSCAN loop
                               for a new instruction)
7E7A   LD   ANDI R10,>0F00     (Change the left color digit. Using
                               ANDI, clear the left digit of the
                               color byte and leave the right digit
                               unchanged)
7E7E        AB R14,R10         (Write the new color byte in R14
                               to R10. It's not shifted first be-
                               cause it's already in the left digit
                               of the left byte)
7E80        JMP AL             (Left color digit updated. Jump
                               back to write the new color byte
                               to the Pattern Descriptor Table)
```

All that remains is to add the name and position of the program to the REF/DEF Table. The following instructions do that:

```
7E82        AORG >701E
701E        DATA >7FE0
7020        AORG >7FE0
7FE0        TEXT 'MULTI '      (Add program name and starting
                               address to the REF/DEF Table.
                               The program, called MULTI, starts
                               at >7D20)
7FE6        DATA >7D20
7FE8        END
```

End the program and select the RUN option of *Mini Memory*. Enter MULTI as the program name and press ENTER. The screen will go black. When you press one of the arrow keys, a small, light red 4 × 4 pixel box appears in the center of the screen. Control the movement of this box, and its resulting trail, with the arrow keys. Pressing the 1 key changes the color of the screen. Continue pressing it until the screen is the color you want. Hitting the 2 key changes the color of the line you're drawing. Press it until the block has the desired color. Pressing the 3 key erases the drawing on the screen and lets you start over. (This is done by sending program control back to >7D64, where the Color Table is made transparent and the initial positions of the block and colors are set.) If you want to draw the blocks slower or faster, change the value in the delay loop, loaded at address >7D9C, to another value.

You can stop the program by pressing FCTN = (QUIT), which has been enabled.

Using the Bitmap Graphics Mode

When the computer is set to display in bitmap graphics mode, each pixel row of a character can have its own foreground and background color, and you can define each of the screen's 768 characters independently. Sprites can be used in this mode, but you cannot use automatic sprite motion. The bitmap mode can be used only on the TI-99/4A computer, not the earlier TI-99/4.

Three tables are used to describe the bitmap graphics screen. One is the Screen Image Table, which is 768 bytes long. Each byte contains the number of the pattern in the Pattern Descriptor Table to be placed at that location. Since the maximum value that can be represented by a byte is 255 and the screen is formed by 768 bytes, it's divided into three sections of 256 bytes each (0–255). Thus, the first 256 bytes in the Screen Image Table represent the first 256 entries in the Pattern Descriptor Table, the second 256 bytes represent the second 256 entries, and the third 256 bytes on the screen represent the third group of 256 bytes in the Pattern Image Table.

In other words, the patterns of each of the screen's 768 positions are stored in the Pattern Descriptor Table. Keeping in mind the fact that each screen location's character definition is eight bytes long, the Pattern Descriptor Table takes up 6144 bytes (768 × 8).

The third table in bitmap mode is the Color Table, which keeps track of the colors of each of the 768 characters on the screen. We mentioned before that each pixel row of a character can have its own foreground and background color. This means that eight bytes are needed to describe the color of one eight-pixel row. The left digit of the byte tells you the foreground color of the pixel row, and the right digit the background color. This table, as the previous two, is divided into three groups of 256 entries each. The first 256 entries represent the colors of the first 256 characters on the screen, the second group the next 256 bytes, and so on.

Locating the Tables

As you might have already noticed, the bitmap graphics mode consumes a lot of VDP memory. When you're preparing to use this mode, you must place all data tables in convenient memory areas so that they won't overlap. The best starting location for the Screen Image Table is at address >1800. The following lines put it there (again, just the instructions are listed below, not the addresses to place them at):

```
LI R0,>0206
BLWP @>6034
```

The Pattern Descriptor Table is usually placed so that it starts at address >0000 by entering the following lines. (Note: If you have the *Editor/Assembler* manual, the values recommended for write-only registers 3 and 4 have been changed in the Addendum.)

```
LI R0,>0403
BLWP @6034
```

The Color Table can be located starting at address >2000 with:

```
LI R0,>03FF
BLWP @>6034
```

The Sprite Attribute List should also be moved to another memory address so that it won't interfere with the bitmap graphics screen. Do this by changing the value in VDP register 5. You can load the Sprite Attribute List at address >1B00, right after the Screen Image Table, by loading VDP register 5 with the value of >36.

```
LI R0,>0536
BLWP @>6034
```

When you correctly place these VDP tables, they'll appear in memory like this.

VDP Tables

>0	**Pattern Descriptor Table**	**6144 bytes**
>1800	**Screen Image Table**	**768 bytes**
>1B00	**Sprite Attribute List**	**1280 bytes**
>2000	**Color Table**	**6144 bytes**

Finally, you should disable sprites which you won't be using. Write the value >D0 to the first address of the Sprite Attribute List to disable *all* sprites. Otherwise, write it to the first byte of the entry of the first *unused* sprite. If you've already moved the list, it will begin at address >1B00.

To disable all sprites, you would then use:

```
LI R0,>1B00
LI R1,>D000
BLWP @>6024
```

Set the Bitmap Graphics Screen

Here, then, are the steps you need to go through when you set the bitmap graphics mode. It's a good idea to actually go through this by entering the instructions on the computer, even though the addresses are not listed below. It'll be good practice.

Step 1. Set bit 6 of VDP register 0 by writing the value >02 to it:

```
LI R0,>0002
BLWP @>6034
```

Step 2. Move the Screen Image Table to the starting address >1800 by loading VDP register 2 with the value >06:

```
LI R0,>0206
BLWP @>6034
```

Step 3. Move the Color Table so that it starts at >2000 by loading VDP register 3 with the value >FF:

```
LI R0,>03FF
BLWP @>6034
```

Step 4. Move the Pattern Descriptor Table to starting address >0000 by loading >03 into VDP register 4:

```
LI R0,>0403
BLWP @>6034
```

Step 5. Move the Sprite Attribute List to another memory area, such as after the Screen Image Table at >1B00, by loading VDP register 5 with >36:

```
LI R0,>0536
BLWP @>6034
```

Step 6. Disable the unused sprites. If *all* sprites are to be disabled, enter:

```
LI R0,>1B00
LI R1,>D000
BLWP @>6024
```

Step 7. Initialize the Screen Image Table by writing the values 0–255 three times, with the following instructions (remember that the Screen Image Table starts at >1800):

```
    LI R0,>1800        (Start writing at >1800)
    CLR R2             (R2 will keep track of the number of
                       times the 256-byte segment has been
                       written)
NQ  CLR R1             (Start writing a zero on the screen)
LP  BLWP @>6024        (Write the value to the Screen Image
                       Table)
    INC R0             (Increase screen position to be written)
    AI R1,>0100        (Increase the value to be written on the
                       screen by one)
    CI R1,>0000        (Past >FF, meaning that the last
                       character has been written?)
    JNE LP             (If not, stay in the printing loop)
    INC R2             (Increase the number of 256-byte seg-
                       ments written)
```

```
     CI R2,3                  (Have all three 256-byte groups been
                              written?)
     JNE NQ                   (If not, repeat the printing loop)
```

Step 8. Clear the Pattern Descriptor Table and the Color Table before using the bitmap graphics screen. This is done by making the colors of the Color Table transparent (>00) and all character definitions zero with the VSBW utility. Both tables are >1800 bytes long. The Pattern Descriptor Table starts at >0000 and the Color Table at >2000. You can then enter

```
     CLR R0
     CLR R1
LP   BLWP @>6024
     INC R0
     CI R0,>1800
     JNE LP
```

to clear the Pattern Descriptor Table. A similar loop to clear the Color Table, which starts at >2000 and extends to >3800 (excluded), would look like this:

```
     LI R0,>2000
     CLR R1
L2   BLWP @>6024
     INC R0
     CI R0,>3800
     JNE L2
```

Creating High-Resolution Graphics

To draw on the bitmap screen once it's been initialized, you must find the byte in the Pattern Descriptor Table which holds the pixel you want to set, the bit in that byte you have to set, and the position of the color byte for that pixel in the Color Table.

Here's the method to find the correct position in the Pattern Descriptor Table and Color Table of the bytes which have to be updated: Assume that the variable X contains the column position of the pixel and Y contains the row position. First of all, divide the row and column values by eight. Imagine QX as the result and RX the remainder of the division of X, and QY and RY as the result and remainder of the division of Y.

The value of variable QY tells you how many complete 32-character rows there are before the line which contains the character you want to change. The value of QX indicates how many complete 8 × 8 pixel characters come before the character you want to alter.

The location of the character byte in the Pattern Descriptor Table is thus given by the formula:

BT = 32*QY+QX+RY

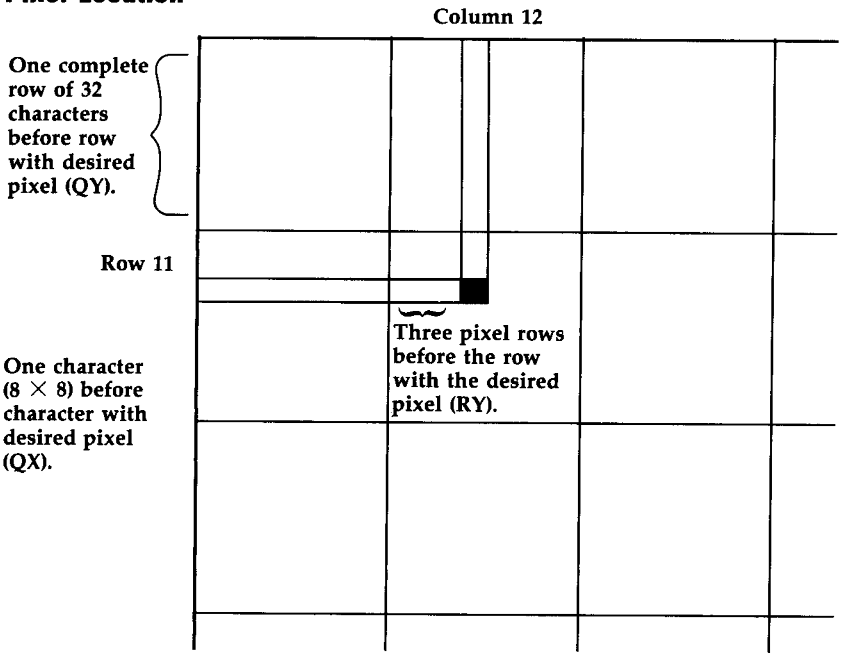
where BT is the byte's location. RY is added in the formula because it tells you how many pixel rows (or bytes) exist before the pixel row holding the character you're going to change.

For example, if the bit to be changed is in row 11 and column 12 of the screen, QX and RX would be 1 and 4 respectively (12÷8 gives a result of 1 with remainder of 4). QY and RY would be 1 and 3 (11÷8 gives a result of 1 with a remainder of 3). The formula would then be:

BT = 32*1+1+3

The result is 36. Take a look at the figure below for an illustration of exactly where the pixel is located, and how the values of QX, RX, QY, and RY locate the byte.

Pixel Location


Column 12
One complete row of 32 characters before row with desired pixel (QY).
Row 11
One character (8 × 8) before character with desired pixel (QX).
Three pixel rows before the row with the desired pixel (RY).

RX (the remainder of the division of the column number by eight) determines the bit of the byte to turn on. Since there are eight bits, this number ranges from 0 to 7. If the remainder is 0, the leftmost pixel of the eight-pixel row must be set by setting the leftmost bit. If the remainder is 7, the rightmost bit must be set. The bits to set for each possible value of RX are:

Remainder (RX)	**Bit to be set**	**Byte value**
0	10000000	>80
1	01000000	>40
2	00100000	>20
3	00010000	>10
4	00001000	>08
5	00000100	>04
6	00000010	>02
7	00000001	>01

To determine what byte value to use, you can add all eight possible values in memory starting at a labeled address:

```
TX DATA >8040,>2010,>0804,>0201
```

Then, if you load R7, for instance, with the starting position of this data, and add the value of the remainder previously loaded in another register (R5, for example), you'll have the correct byte.

If R5 is 0, byte >80 is arrived at; if the remainder is 1, value >40; and so on. Then, by using the SOCB instruction, you can set the corresponding byte read from the table. The byte with the new pixel set is rewritten to the Pattern Descriptor Table.

Once the bit has been set, you must add the color of the byte which contains the modified bit in the Color Table. Of course, you have to locate the byte in the Color Table which stores the color of the changed byte. Because the starting address of the Color Table is exactly >2000 bytes greater than the Pattern Descriptor Table, all you have to do is add >2000 to the Pattern Descriptor Table entry address to arrive at the correct position in the Color Table.

The color for each eight-pixel row is assigned by a hexadecimal byte. The byte's left digit assigns the color of the pixels that are *set* and the right digit sets the color of *reset* pixels. Write this byte to the Color Table and the pixel row is assigned a color.

A short assembly language routine on page 336 of the *Editor/Assembler* manual allows you to calculate the byte to be changed in the Pattern Descriptor Table. The routine is a short method to perform the calculations mentioned before, without having to use the DIV instruction. R0 must be loaded with the column value of the pixel to be set, and R1 with the row value. At the end of the routine, R4 is loaded with the position in the Pattern Descriptor Table of the byte, and R5 with the remainder of the division of the column number.

This is the routine, reproduced courtesy of Texas Instruments, Incorporated.

```
MOV R1,R4
SLA R4,5
SOC R1,R4
ANDI R4,>FF07
MOV R0,R5
ANDI R5,7
A R0,R4
S R5,R4
```

The corresponding position in the Color Table is found by adding >2000 to the value in R4. To set R7 with the correct bit, use the following instructions, as earlier explained:

```
    LI R7,TX
    A R5,R7
    SOCB *R7,R1         (Where R1 is the old value byte read
                        from the Pattern Descriptor Table)
     .
     .
     .
TX  DATA >8040,>2010,>0804,>0201
```

Hi-Res Bouncer

This bitmap mode program draws a diagonal line on the screen. When the line encounters a screen edge, it changes direction. The line continues, eventually creating a pattern on the screen. Each dot position is calculated by adding 1 or −1 to the row and column positions of the pixel, according to the direction of the line.

The first part of the program sets the VDP tables in the appropriate memory areas.

```
7D00          LWPI >70B8          (Load memory area for the
                                   registers)
7D04          LI R0,>0002         (Load VDP register 0 with >02 to
                                   select bitmap mode)
7D08          BLWP @>6034         (VDP Write To Register)
7D0C          LI R0,>0206         (Screen Image Table placed start-
                                   ing at >1800)
7D10          BLWP @>6034         (VDP Write To Register)
7D14          LI R0,>03FF         (Prepare to place the Color Table
                                   starting at >2000)
7D18          BLWP @>6034         (VDP Write To Register)
7D1C          LI R0,>0403         (Prepare to move the Pattern
                                   Descriptor Table to start at >0000)
7D20          BLWP @>6034         (VDP Write To Register)
7D24          LI R0,>0536         (Prepare to move the Sprite
                                   Attribute List to the memory area
                                   beginning at >1B00)
7D2C          LI R0,>0701         (Screen will be black)
7D30          BLWP @>6034         (VDP Write To Register)
```

The Screen Image Table is initialized by numbering it three times (0–255).

```
7D34          LI R0,>1800         (Screen Image Table starts at
                                   >1800)
7D38          CLR R2              (R2 will keep track of the times
                                   the screen has been numbered
                                   0–255)
7D3A   L1     CLR R1              (Value to be written to the screen
                                   address)
7D3C   L2     BLWP @>6024         (Print the character on the screen)
7D40          INC R0              (Increase screen printing position)
7D42          AI R1,>0100         (Increase code of character to be
                                   printed)
7D46          CI R1,0             (Have the 256 characters been
                                   printed?)
7D4A          JNE L2              (If not, stay in loop L2)
7D4C          INC R2              (Increase number of 256-byte seg-
                                   ments that have been printed)
7D4E          CI R2,3             (All three groups printed?)
7D52          JNE L1              (If not, stay in the initializing
                                   loop)
```

Then the Pattern Descriptor Table and Color Table are cleared.

7D54		CLR R0	(Pattern Descriptor Table starts at position 0)
7D56		CLR R1	(Code of character to be printed is zero)
7D58	L3	BLWP @>6024	(Write the character to the Pattern Descriptor Table)
7D5C		INC R0	(Increase the position in the table)
7D5E		CI R0,>1800	(Has the last table position been reached?)
7D62		JNE L3	(If not, stay in loop L3)
7D64		LI R0,>2000	(Color Table begins at address >2000)
7D68		CLR R1	(Value to be written to the table is zero)
7D6A	L4	BLWP @>6024	(Print the value to the table)
7D6E		INC R0	(Increase table position)
7D70		CI R0,>3800	(Has the last position of the table been reached?)
7D74		JNE L4	(If not, stay in loop L4)

Sprites are then disabled and the initial register values are set to indicate initial position and initial direction.

7D76		LI R0,>1B00	(Sprite Attribute List starts at address >1B00)
7D7A		LI R1,>D000	(Value to be written to disable sprites is >D0)
7D7E		BLWP @>6024	(Write the value to the Sprite Attribute List)
7D82		LI R8,128	(Load column of the initial pixel)
7D86		LI R9,95	(Load row of initial pixel)
7D8A		LI R14,1	(Horizontal motion: right)
7D8E		LI R15,1	(Vertical motion: down)

The main execution loop, which computes the position of the new pixel to be set, is then added.

7D92	LP	A R14,R8	(Move printing position one pixel left or right, according to the value of R14)

7D94	A R15,R9	(Move printing position one pixel down or up, according to the value of R15)
7D96	CI R8,256	(Has the right-hand column been reached?)
7D9A	JLT $+4	(If not, jump to the next check)
7D9C	NEG R14	(It has. Invert the horizontal direction)
7D9E	CI R8,0	(Has the leftmost column been reached?)
7DA2	JGT $+4	(If not, jump to the third check)
7DA4	NEG R14	(Invert the horizontal movement direction)
7DA6	CI R9,191	(Has the bottom pixel row been reached?)
7DAA	JLT $+4	(If not, jump to the fourth and last check)
7DAC	NEG R15	(Invert vertical movement direction)
7DAE	CI R9,0	(Has the top pixel row been reached?)
7DB2	JGT $+4	(If not, skip the following updating instruction)
7DB4	NEG R15	(Invert vertical movement direction)

The next segment calculates the byte in the Pattern Descriptor Table to be changed and indicates the bit which must be set in that byte.

7DB6	MOV R9,R4	(Copy the row value of the pixel in R4)
7DB8	SLA R4,5	(Multiply this value by 32 by shifting every bit five positions left)
7DBA	SOC R9,R4	(Set the bits in R4 which are also set in R9)
7DBC	ANDI R4,>FF07	(Keep the left byte of R4 and the three right bits of the right byte of R4)
7DC0	MOV R8,R5	(Copy the column value of the pixel in R5)

7DC2	ANDI R5,7	(Keep the three right bits of the word. R5 holds the remainder of the division by eight)
7DC6	A R8,R4	(Add the column value of the pixel to be set to the value in R4)
7DC8	S R5,R4	(Subtract the value in R5 from the value in R4. The byte's position in the Pattern Descriptor Table is stored in R4 and the remainder of the division is stored in R5)

The byte to be changed from the Pattern Descriptor Table has to be read next. The SOCB instruction is used to set the bit representing the pixel in the byte read from the table. Once this bit has been set, the program writes the updated byte back to the Pattern Descriptor Table.

7DCA	MOV R4,R0	(Move position of the byte to be read to R0 for the VSBR utility)
7DCC	BLWP @>602C	(Read the byte from the table)
7DD0	LI R7,TX	(Eight value bytes are located starting at label TX. One of them represents the byte with the desired bit set)
7DD4	A R5,R7	(Add the remainder. This locates the correct value byte)
7DD6	SOCB *R7,R1	(Set the bit of this value byte, located at the address stored in R7, in the byte read from the Pattern Descriptor Table)
7DD8	BLWP @>6024	(Write the byte back to the Pattern Descriptor Table to the same address it was read from)
7DDC	AI R0,>2000	(Find the corresponding position in the Color Table, >2000 bytes ahead)
7DE0	LI R1,>A000	(Load the color byte to be written to that address: dark yellow [>A] on transparent [>0])
7DE4	BLWP @>6024	(Write the byte to the Color Table)
7DE8	JMP LP	(Jump back to the main control loop)

The last section of the program adds the DATA containing the eight value bytes. The name and position of the program are also added to the REF/DEF Table.

```
7DEA  TX  DATA >8040,>2010,>0804,>0201
7DF2      AORG >701E
701E      DATA >7FE0
7020      AORG >7FE0
7FE0      TEXT 'BOUNCE'
7FE6      DATA >7D00
7FE8      END
```

Now you can use the RUN option of the *Mini Memory* cartridge to run the program. Just type in BOUNCE and press ENTER. You'll see the screen initialization taking place, the screen cleared, and the program begin. (The next program, "Hi-Res Draw," sets the computer in text mode while the screen initialization takes place, so it's not visible. There's usually more than one way to create a program.)

Artist's Sketchpad

This next program, Hi-Res Draw, allows you to draw high-resolution pictures on the bitmap graphics screen. The lines can be drawn in any of eight directions, screen color and drawing colors can be changed, and drawing speed can be set slow or fast. A frame can even be drawn to surround your graphics if you wish. The keys used to control the program's features are:

Key	**Function**
S	Left
D	Right
X	Down
E	Up
R	Up and right
W	Up and left
Z	Down and left
C	Down and right
1	Changes the screen color each time it is pressed
2	Changes pixel color each time it is pressed
3	Slow drawing speed

4	Fast drawing speed
K	Clears screen, resets initial position and initial colors
F	Draws a frame around the screen

This is a long program, one of the longest in this book. Enter each segment carefully. An error may mean having to retype the whole program. Make sure to frequently recheck the memory address shown on the screen with the address in the program listing.

As you're entering this program, you'll suddenly return to the *Line-by-Line Assembler* title screen and previous instructions you've written will appear. Ignore these and continue entering Hi-Res Draw. (See the section "The Screen Buffer" in Chapter 12 for more information on this phenomenon.)

Hi-Res Draw

The first section places you in the correct area of memory to begin the program, leaving plenty of space for the labels used, sets the screen to black, and puts the computer in text mode so the bitmap mode initialization cannot be seen. Once the bitmap mode has been set, the text screen is canceled.

7D00	**AORG >7D30**	(Program will start at >7D30)
7D30	**LWPI >70B8**	(Load memory area for the registers)
7D34	**CLR @>8374**	(A standard keyboard scan will be executed later in the program, so clear address >8374)
7D38	**LI R0,>0711**	(Prepare to write >11 to VDP register 7)
7D3C	**BLWP @>6034**	(VWTR : screen color set to black)
7D40	**LI R0,>01F0**	(Prepare to write <F0 to VDP register 1)
7D44	**BLWP @>6034**	(VWTR : text mode selection)

Locate the tables correctly for bitmap mode and disable the sprites.

7D48	**LI R0,>0206**	(Prepare to write >06 to VDP register 2)
7D4C	**BLWP @>6034**	(VWTR: Screen Image Table located at >1800)

```
7D50        LI R0,>03FF     (Prepare to write >FF to VDP
                            register 3)
7D54        BLWP @>6034     (VWTR: Color Table located start-
                            ing at >2000)
7D58        LI R0,>0403     (Prepare to write >03 to VDP reg-
                            ister 4)
7D5C        BLWP @>6034     (VWTR: Pattern Descriptor Table
                            located starting at >0000)
7D60        LI R0,>0536     (>36 will be written to VDP reg-
                            ister 5)
7D64        BLWP @>6034     (Sprite Attribute List located
                            beginning at >1B00)
7D68        LI R0,>1B00     (To disable sprites, write >D0 to
                            >1B00)
7D6C        LI R1,>D000     (Value to be written is >D0)
7D70        BLWP @>6024     (Write the value to the specified
                            address)
```

The next step is to initialize the Screen Image Table, which is located starting at >1800, by writing the values 0–255 three times on it.

```
7D74        LI R0,>1800     (Screen Image Table starts at
                            >1800)
7D78        CLR R2          (R2 will keep count of the number
                            of 255-byte segments already
                            written on the screen)
7D7A   NQ   CLR R1          (Initial value to be written is >00)
7D7C   L1   BLWP @>6024     (Print the value to the table)
7D80        INC R0          (Increase screen printing position)
7D82        AI R1,>0100     (Increase value to be written)
7D86        CI R1,>0000     (Have the 256 values been
                            printed?)
7D8A        JNE L1          (If not, stay in loop L1)
7D8C        INC R2          (Increase number of 255-byte seg-
                            ments written)
7D8E        CI R2,3         (Have all three segments been
                            written?)
7D92        JNE NQ          (If not, return to repeat the print-
                            ing sequence)
```

Now the Pattern Descriptor Table and Color Table will be cleared by writing >00 to all locations. Both tables are >1800 bytes long and are located at >0000 and >2000 respectively.

```
7D94  CS  CLR R0          (Pattern Descriptor Table begins at
                          address >0000)
7D96      CLR R1          (Zero to be written to all
                          addresses)
7D98  L2  BLWP @>6024     (Write the value to the table)
7D9C      INC R0          (Increase position in the table)
7D9E      CI R0,>1800     (Has the end of the table been
                          reached?)
7DA2      JNE L2          (If not, stay in the clearing loop)
7DA4      LI R0,>2000     (Color Table starts at >2000)
7DA8  L3  BLWP @>6024     (Write >00 to the Color Table
                          address)
7DAC      INC R0          (Increase table position)
7DAE      CI R0,>3800     (Has the last position been
                          passed?)
7DB2      JNE L3          (If not, table still not completely
                          cleared. Stay in loop L3)
```

Now that the VDP tables have been located and initialized, the computer is set in bitmap mode and the text screen mode is canceled. The coordinates of the initial pixel will be set, together with the initial drawing and screen color. The delay loop value for the drawing speed will be set, and the program then branches to the pixel drawing routine to display the initial pixel on the screen.

```
7DB4      LI R0,>0002     (Prepare to write >02 to VDP reg-
                          ister 0)
7DB8      BLWP @>6034     (VWTR: select bitmap mode)
7DBC      LI R0,>01E0     (Prepare to write >E0 to VDP reg-
                          ister 1)
7DC0      BLWP @>6034     (VWTR: cancel text mode)
7DC4      LI R3,128       (Column value of the initial pixel)
7DC8      LI R4,96        (Row value of the initial pixel)
7DCC      LI R5,>0001     (Initial screen color is black [>01])
7DD0      LI R6,>A000     (Initial pixel color is dark yellow
                          on transparent [>A0])
```

```
7DD4        LI R15,5000        (Initial delay loop value [drawing
                               speed] is 5000)
7DD8        BL @DR             (Branch to the DR subroutine to
                               set the initial pixel)
```

All the initial conditions have been set. The next segment enables and disables VDP interrupts so the program can be stopped with FCTN = (QUIT), executes the delay loop to set the pixel drawing speed, and scans the keyboard, moving the ASCII code of the key pressed to R1 for the corresponding checks.

```
7DDC  LP    LIMI 2             (Enable VDP interrupts to allow
                               FCTN =(QUIT))
7DE0        LIMI 0             (Disable VDP interrupts)
7DE4        MOV R15,R2         (Move the delay value to R2)
7DE6        DEC R2             (Decrease delay value)
7DE8        JNE $-2            (If not zero, stay in delay loop)
7DEA        BLWP @>6020        (Branch to scan the keyboard)
7DEE        CLR R1             (Prepare R1 to receive the ASCII
                               code of the key pressed)
7DF0        MOV @>8375,R1      (Move the ASCII code of the key
                               pressed to the right byte of R1)
```

Once the ASCII code of the key pressed has been moved to R1, the program checks and updates the values accordingly. It first checks to see if one of the four arrow keys (up, down, right, or left) was pressed, and if one was, updates the pixel coordinates.

```
7DF4        CI R1,69           (Was the up-arrow key [E]
                               pressed?)
7DF8        JNE $+6            (If not, jump six bytes ahead to
                               the next check)
7DFA        DEC R4             (Decrease the row number of the
                               pixel)
7DFC        JMP CK             (Jump to the checking routine at
                               CK. There the pixel value is
                               checked to see if it's in screen
                               limits)
7DFE        CI R1,88           (Was the down arrow [X]
                               pressed?)
```

7E02	JNE $+6	(If not, jump six bytes ahead to the next check)
7E04	INC R4	(Increase the row number of the pixel)
7E06	JMP CK	(Jump to the screen limits check)
7E08	CI R1,68	(Was the right arrow [D] pressed?)
7E0C	JNE $+6	(If not, jump to the next check)
7E0E	INC R3	(Increase column coordinate of the pixel)
7E10	JMP CK	(Jump to the screen limits check)
7E12	CI R1,83	(Was the left arrow [S] pressed?)
7E16	JNE $+6	(If not, jump to the next check, six bytes ahead)
7E18	DEC R3	(Decrease column value of the pixel)
7E1A	JMP CK	(Jump to check if the pixel is in screen limits)

Now the comparisons for the four diagonal directions will be coded:

7E1C	CI R1,82	(Was the R [up and right] key pressed?)
7E20	JNE $+8	(If not, jump to the next comparison, eight bytes ahead)
7E22	DEC R4	(Decrease the row value of the pixel)
7E24	INC R3	(Increase the column value of the pixel)
7E26	JMP CK	(Jump to check if the pixel is still within screen limits)
7E28	CI R1,87	(Was the W [up and left] key pressed?)
7E2C	JNE $+8	(If not, jump to the next check)
7E2E	DEC R4	(Decrease row value of the pixel)
7E30	DEC R3	(Decrease column value of the pixel)
7E32	JMP CK	(Jump to screen limit check)
7E34	CI R1,90	(Was the Z [down and left] key pressed?)
7E38	JNE $+8	(If not, jump to the next check)
7E3A	INC R4	(Increase row value of the pixel)

7E3C	DEC R3	(Decrease column value of the pixel)
7E3E	JMP CK	(Jump to check if the pixel is in screen limits)
7E40	CI R1,67	(Was the C [down and right] key pressed?)
7E44	JNE $+8	(Jump eight bytes to the next check if not)
7E46	INC R4	(Increase row value of the pixel)
7E48	INC R3	(Increase column value of the pixel)
7E4A	JMP CK	(Jump to screen limit check)

The keys which change color and speed, clear the screen, or draw the frame have to be checked.

7E4C	CI R1,75	(Was the K pressed to clear the screen?)
7E50	JEQ CS	(If it was, jump back to the color initialization routine [clearing the Pattern Descriptor Table and the Color Table] in the beginning of the program)
7E52	CI R1,49	(Was the 1 key [change screen color] pressed?)
7E56	JEQ SC	(If it was, jump to the routine at SC, where the screen color will be changed)
7E58	CI R1,50	(Was the 2 key [change pixel color] pressed?)
7E5C	JEQ LC	(If it was, jump to routine at label LC, where the pixel color will be updated)
7E5E	CI R1,51	(Was the 3 key [slow drawing speed] pressed?)
7E62	JEQ DS	(If it was, jump to label DS, where the speed will be set to slow by changing the delay loop value)
7E64	CI R1,52	(Was the 4 key [fast drawing speed] pressed?)
7E68	JEQ DF	(If it was, jump to DF where the speed is increased by changing the value of the delay loop)

```
7E6A        CI R1,70        (Was the F key [draw frame]
                            pressed?)
7E6E        JEQ FR          (If it was, jump to the routine to
                            draw the frame at FR)
7E70        JMP LP          (Ignore any other key by return-
                            ing to the KSCAN loop)
```

The next segment is the routine to change the screen color each time the 1 key is pressed.

```
7E72   SC   CI R5,>000F     (At the last screen color—white?)
7E76        JNE $+4         (If not, skip the instruction to re-
                            set the screen color to black)
7E78        CLR R5          (Make screen color transparent.
                            When the screen color is next up-
                            dated, it will be black)
7E7A        INC R5          (Update screen color by adding
                            one to the color code)
7E7C        MOV R5,R0       (Move the color byte for the
                            VWTR utility)
7E7E        AI R0,>0700     (Write >07 to the left byte of R0,
                            where the color is set. >07 is used
                            because the screen color byte has
                            to be written to VDP register 7)
7E82        BLWP @>6034     (VWTR utility: screen color is
                            changed)
7E86        LI R2,20000     (Delay value so the color change
                            can be seen)
7E8A        DEC R2          (Decrease value of the delay loop
                            in R2)
7E8C        JNE $-2         (If the delay is not over, stay in
                            the loop)
7E8E        JMP LP          (Screen color changed. Return to
                            the main loop)
```

The pixel color is changed when the 2 key is pressed by the following segment.

```
7E90   LC   AI R6,>1000     (Update pixel color by adding one
                            to the current color)
7E94        LI R2,20000     (Delay to give the user time to re-
                            lease the key, so the pixel color
                            will not change too quickly)
7E98        DEC R2          (Decrease delay value in R2)
```

```
7E9A          JNE $-2          (If not zero, stay in the delay
                               loop)
7E9C          JMP DR           (Jump to the drawing routine,
                               where the pixel color will be
                               updated)
```

The routines to set the pixel's drawing speed are included in the next section.

```
7E9E    DS    LI R15,5000      (For slow speed, load a delay
                               value of 5000 in R15)
7EA2          JMP LP           (Slow speed set. Return to main
                               control loop, LP)
7EA4    DF    LI R15,600       (For fast speed, load a delay value
                               of 600 in R15)
7EA8          JMP LP           (Fast speed set. Return to main
                               control loop, LP)
```

This routine draws a frame when the F key is pressed. Four loops are used to draw the four edges.

```
7EAA    FR    MOV R3,R9        (Store the current pixel column in
                               R9 while the frame is being
                               drawn)
7EAC          MOV R4,R10       (Store current pixel row in R10
                               while the frame is being drawn)
7EAE          CLR R3           (Start drawing the frame at col-
                               umn 0)
7EB0          CLR R4           (Start drawing the frame at row 0)
7EB2    F1    BL @DR           (Loop to draw the top edge.
                               Branch to set the pixel)
7EB6          INC R3           (Increase column value)
7EB8          CI R3,256        (Has the right side of the screen
                               been reached—top edge finished?)
7EBC          JLT F1           (If still not finished drawing the
                               top edge, stay in loop L1)
7EBE          DEC R3           (Right edge drawn in column 255)
7EC0    F2    BL @DR           (Set the pixel by branching to the
                               subroutine DR)
7EC4          INC R4           (Increase the row of the pixel)
7EC6          CI R4,192        (Has the last row been reached?)
7ECA          JLT F2           (If not, stay in the drawing loop
                               F2)
```

```
7ECC        DEC R4          (Set R4 ready, to start drawing the
                            bottom edge)
7ECE   F3   BL @DR          (Branch to DR and set the pixel
                            on the screen)
7ED2        DEC R3          (Decrease the column value of the
                            pixel)
7ED4        JGT F3          (Stay in the drawing loop of the
                            bottom edge as long as column 0
                            is not reached)
7ED6   F4   BL @DR          (Start the fourth loop, to draw the
                            left edge)
7EDA        DEC R4          (Decrease row value of the pixel)
7EDC        JGT F4          (If not zero, stay in drawing loop
                            L4)
7EDE        MOV R9,R3       (Frame finished. Move position of
                            the drawing pixel, stored in R9,
                            back to R3)
7EE0        MOV R10,R4      (Move the current pixel row back
                            to R4)
7EE2        B @LP           (Return to main control loop. The
                            B instruction is used because the
                            address where the main loop be-
                            gins is too far away to be reached
                            by a jump-style instruction)
```

The program next checks whether the pixel to be drawn is in screen limits. If it's not, the row and column values of the pixel are adjusted.

```
7EE6   CK   CI R3,256       (Compare the column value to the
                            maximum column value)
7EEA        JLT $+6         (If it's lower, no updating needed.
                            Jump to the next check)
7EEC        LI R3,255       (Update column value to maxi-
                            mum column value)
7EF0        CI R3,0         (Compare the pixel column to the
                            minimum column value)
7EF4        JGT $+6         (If it's greater, no updating
                            needed. Jump to the next check)
7EF6        LI R3,1         (Update column value)
7EFA        CI R4,192       (Compare pixel row to maximum
                            row value)
```

```
7FFE          JLT $+6         (If pixel is in screen limits, skip
                              the updating instruction)
7F00          LI R4,191       (Pixel off the bottom of the
                              screen. Reset its position)
7F04          CI R4,0         (Check if pixel is out the top of
                              the screen)
7F08          JGT $+6         (If updating not necessary, skip
                              the next instruction)
7F0A          LI R4,1         (Update the row value of the
                              pixel)
7F0E          BL @DR          (Branch to execute the subroutine
                              to set the pixel on the screen and
                              assign it a color)
7F12          B @LP           (Pixel on the screen. Return to
                              loop LP)
```

Now the program's ready to calculate the byte from the Pattern Descriptor Table which has to be changed, and the value which will decide which bit of the byte has to be set.

```
7F16    DR    MOV R4,R12      (Move the value in R4 to R12)
7F18          SLA R12,5       (Multiply the value by 32)
7F1A          SOC R4,R12      (Set the bits in R12 that are also
                              set in R4)
7F1C          ANDI R12,>FF07  (Set the bits set in both R12 and
                              >FF07 in a new word, stored in
                              R12)
7F20          MOV R3,R13      (Move the value in R3 to R13)
7F22          ANDI R13,7      (Store only the right three bits of
                              the word in R13)
7F26          A R3,R12        (Add the value in R3 to the value
                              in R12)
7F28          S R13,R12       (Subtract the value in R13 from
                              the value in R12)
```

This gives the position of the byte to be changed in the Pattern Descriptor Table (in R12) and the value to indicate which bit to set (in R13). Now the bit is set, as well as the color of the pixel.

```
7F2A          MOV R12,R0      (Move the position in VDP RAM
                              from which to read a value to R0
                              for the VSBR routine)
```

```
7F2C        BLWP @>602C       (Read the old byte from the Pat-
                              tern Descriptor Table)
7F2E        LI R7,TX          (Load R7 with the initial position
                              in memory of the eight bytes with
                              the possible set bit combinations)
7F34        A R13,R7          (Add the value in R13. Indicates
                              which byte to use)
7F36        SOCB *R7,R1       (Set the bits in the left byte of R1
                              which are also set in the left byte
                              of the word in R7. The new set
                              pixel will be written to the old
                              byte read from the Pattern
                              Descriptor Table)
7F38        BLWP @>6024       (Write the updated byte back to
                              the Pattern Descriptor Table)
7F3C        AI R0,>2000       (Add >2000 to the value in R0 to
                              get the correct address of the byte
                              to change in the Color Table)
7F40        MOV R6,R1         (Move the color byte to R1 for the
                              VSBW utility)
7F42        BLWP @>6024       (Write the color byte to the table)
7F46        B *R11            (Return from the subroutine)
```

Finally, DATA is added to the program and the name and position of the program are placed in the REF/DEF Table.

```
7F48   TX   DATA >8040,>2010,>0804,>0201 (The eight value
                              bytes, with the eight possible
                              pixel combinations)
7F50        AORG >701E
701E        DATA >7FE0
7020        AORG >7FE0
7FE0        TEXT 'BITMAP'
7FE6        DATA >7D00
7FE8        END
```

Run the program, called BITMAP. The screen will be set to black. Wait for a few seconds and a dot will light up in the center of the screen. Use the directional keys to draw lines. When you select F to frame the screen, it's drawn in the current color. To change the frame color, alter the color you're using and hit the F key again. To make the frame disappear, redraw it with transparent color (the first color after white).

You can also make it the screen color, but if you later change the screen, the frame will again be visible.

Though the program starts off with a black screen, drawing on a black screen causes some problems with the colors (vertical lines tend to differ in color). A good combination to try is black lines on a white screen.

You can leave the program pressing FCTN = (QUIT).

Resetting the Graphics Mode

When you change the graphics mode on the computer and want to return to the normal graphics mode used by BASIC, bit 6 of VDP register 0 and bits 3 and 4 of VDP register 1 must be reset. This can be done with the following lines. (Use the appropriate instructions according to the VDP register you want to reset.)

```
LI R0,>0000
BLWP @>6034        (Reset bitmap mode)
LI R0,>01E0
BLWP @>6034        (Reset text and multicolor mode)
```

Also, any tables you've moved in memory, particularly if you've been using the bitmap mode, should be returned to their normal memory areas.

The following example program displays the message HELLO on the screen and then writes the value of >02 to register 0 to set the computer in bitmap mode. When this is done, no text will be visible on the screen. The keyboard will be scanned. When a key is detected, byte >00 is written to VDP register 0, thus returning the computer to graphics mode and making the text readable on the screen.

Bitmap Back to Graphics

```
7D00    LWPI >70B8       (Load memory area so program
                         can be run with EASY BUG)
7D04    LI R0,300        (Screen position where the text
                         must be displayed)
7D08    LI R1,TX         (Position in memory of the text to
                         be displayed)
7D0C    LI R2,5          (Length of the text)
7D10    BLWP @>6028      (Display the text on the screen)
7D14    LI R0,>0002      (Prepare to write >02 to VDP reg-
                         ister 0)
```

```
7D18        BLWP @>6034     (VWTR: set the computer in
                            bitmap mode)
7D1C        CLR @>8374      (Standard keyboard scan to be
                            executed)
7D20   LP   BLWP @>6020     (Branch to scan the keyboard)
7D24        MOV @>8375,R1   (Move the ASCII code of the key
                            pressed to the right byte of R1)
7D28        CI R1,255       (Compare the code to >FF [255]:
                            no key pressed)
7D2C        JEQ LP          (If no key was pressed, stay in the
                            KSCAN loop)
7D2E        LI R0,>0000     (Key pressed. Prepare to reset the
                            computer to graphics mode)
7D32        BLWP @>6034     (Write >00 to VDP register 0.
                            Graphics mode selection)
7D36        JMP $           (Stop the program with an endless
                            loop)
7D38   TX   TEXT 'HELLO'    (Text to be displayed)
```

Type END and then run the program from EASY BUG. After you type E7D00, barely touch the ENTER key. If you press it too long, the mode switching effect will be obscured.

One More Chapter

After countless examples and illustrations, you've learned how to program in assembly language. You've created simple routines and you've created complex programs that turn your TI into a artist's hi-res sketchpad.

Chapter 12, the next (and last) chapter in this book, offers a wide selection of hints and tips you'll find extremely useful as you program in assembly language. From using the limited and valuable memory of the *Mini Memory* cartridge to displaying the value in a register, this section includes numerous techniques that will help you develop your programming skills.

Chapter 12

Assembly Language Programming Techniques

Assembly Language Programming Techniques

Programming in assembly language is easy once you learn how to handle the instructions, directives, and general syntax and operation of the assembler you're using. Until you have this well in hand, it's best to stay with short and simple assembly language routines. Try to avoid long and complicated programs at first.

An easy way to become more comfortable with assembly language, and to work towards your goal of creating more complex programs is to experiment and test the subroutines in your collection. That's the primary purpose of the examples in this book. Once you've mastered them, they can easily be inserted in a larger program.

When you feel ready to tackle a large program, it will help if you can divide it into segments and test each one individually. Locating errors won't be as difficult. And remember: If your program is long and uses several labels, leave enough memory for the Symbol Table.

Debugging

If your program does not work as you expected, and you need to know what values are loaded in each of your registers, you can use the EASY BUG M command. Select EASY BUG. Assuming that the memory area for the work registers was loaded beginning at >70B8, you'd type:

M70B8

immediately after the ? prompt, and then press ENTER. The values stored in >70B8 and >70B9 are the contents of R0. The values in >70BA and >70BB are the contents of R1, and so on. Remember that using EASY BUG changes the values of some of the registers.

With EASY BUG, you can check and even change any value in CPU RAM. To check or modify a value in VDP RAM, such as values in data tables, use the V command from EASY BUG.

By the way, you don't need to include the *R* to indicate you're using a register while programming. In other words,

LI R3,5 and **LI 3,5**

mean the same thing to the computer. It knows when a value is meant as a number or when it's meant as a register. The registers in the example programs in this book have all been specified with the *R* simply to avoid confusion.

Saving Memory

When you want to write an assembly language program and feel you won't have enough memory for it, the best solution is to create all the sections which don't directly affect the program's execution speed (character definitions, colors, title, and options) in BASIC, and then link them to the assembly language program. This will save a considerable amount of memory.

Loading Your Program from BASIC

An assembly language program can be loaded into the *Mini Memory* module or the memory expansion entirely from BASIC. You can use BASIC's LOAD subroutine to load sections of the assembly language program to CPU RAM and BASIC's POKEV subroutine to load the values to be written to VDP RAM. Letting BASIC POKE the assembly language program into memory allows a BASIC and assembly language program to be loaded directly with the normal BASIC procedure. You won't need EASY BUG's L command to load the assembly language program separately.

The BASIC program POKEs the machine language program to memory. But how is it done?

An assembly language program, once it's assembled, is a list of binary values which are displayed in hexadecimal notation. These instructions can be POKEd directly into memory from BASIC. For example, in assembly language, the instruction *B *R11* is translated to hexadecimal as:

>045B

But >04 is 4 in decimal and >5B is 91. If you load the memory address where you want the instruction to be written with the values 4 and 91, you would, in reality, be entering *B *R11*:

CALL LOAD (*xxxxx*,4,91) (Places the instruction *B *R11* at the memory address *xxxxx*)

A BASIC program that places an assembly language program in memory is called a *BASIC loader.*

To load a complete assembly language program into memory from BASIC, you must load its decimal equivalents one byte at a time. Once the program has been loaded into memory, you must update the LFAM (Last Free Address of the Module) at >701E (updating >701C is unnecessary as long as you're sure that your program does not overwrite the entry point of the REF/DEF Table). Load the program name and starting address to the REF/DEF Table just as you've loaded the program to memory so that the program can be called with the LINK subroutine from BASIC.

Let's look at an example. First of all, let's enter an assembly language program using the *Line-by-Line Assembler*:

```
7D00        LI R0,300          (Position onscreen to display text)
7D04        LI R1,TX           (Position of text in CPU RAM)
7D08        LI R2,4            (Length of the text to be
                               displayed)
7D0C        BLWP @>6028        (Display the text on the screen)
7D10        CLR @>837C         (Clear the status byte to avoid
                               false errors upon return to BASIC)
7D14        B *R11             (Return to BASIC)
7D16   TX   DATA >B4A5,>B3B4   (Text to be displayed. It's
                               added with the DATA directive
                               because the screen bias of >60
                               has to be added to the ASCII code
                               of each character)
7D1A        AORG >701E
701E        DATA >7FE0
7020        AORG >7FE0
7FE0        TEXT 'TRYOUT'      (Add the name and address of the
                               program to the REF/DEF Table)
7FE6        DATA >7D00
7FE8        END
```

The things you'll have to load into memory are:

1. The program itself, from >7D00 to >7D19 (32000 to 32025)
2. The updating of the LFAM at >701E–>701F (28702 and 28703)

3. The name and position of the program at the REF/DEF Table (>7FE0 ->7FE7 (32736–32743))

Now that you know where in memory the corresponding values have to be loaded, the following BASIC program will read the decimal values (which later will be POKEd into memory), and then print them.

```
100 CALL CLEAR
110 FOR A=32000 TO 32025
120 CALL PEEK (A,DECVAL)
130 PRINT DECVAL;
140 NEXT A
```

Running this BASIC program would print the values:

```
2  0  1  44  2  1  125  22
2  2  0  4  4  32  96  40
4  224  131  124  4  91
180  165  179  180
```

These decimal values are your assembly language program.

To find the values to load in the address of the LFAM, read the contents of addresses >701E and >701F with:

```
100 CALL PEEK(28702,A,B)
110 PRINT A;B
```

These two lines will print the values *127* and *224*. These are the values to load in the address of the LFAM (location >7FE0).

Finally, to find the values to load the name and position of the program to the REF/DEF Table, you'd use the BASIC lines:

```
100 CALL PEEK(32736,A,B,C,D,E,F,G,H)
110 PRINT A;B;C;D;E;F;G;H
```

which will return

```
84  82  89  79  85  84  125
0
```

These are the values for the name and position of the program, since:

84 = T
82 = R
89 = Y
79 = O
85 = U

```
 84 = T
125 = >7D
  0 = >00
```

Once you have all the values at hand, you can type NEW and enter the BASIC program which loads the assembly language program into memory and links it if desired.

First of all, you need to create a loop to read the decimal values from a DATA statement and load them into memory at the correct addresses. That part of the BASIC loader could be:

```
100 CALL INIT
110 CALL CLEAR
120 FOR A=32000 TO 32025
130 READ DECVAL
140 CALL LOAD(A,DECVAL)
150 NEXT A
160 DATA 2,0,1,44,2,1,125,22,2,2,0,4,4,32
165 DATA 96,40,4,224,131,124,4,91,180,165,179,180
```

The next line loads the LFAM:

```
170 CALL LOAD(28702,127,224)
```

And the name and position of the program is loaded to the REF/DEF Table with this line:

```
180 CALL LOAD(32736,84,82,89,79,85,84,125,0)
```

To link the BASIC program to the assembly language program at any point, these two lines are needed:

```
190 CALL LINK("TRYOUT")
200 END
```

When you run this program, even if the *Mini Memory* cartridge has been reinitialized and has a blank memory, the program will directly load the assembly language program, the LFAM, and the REF/DEF Table entry, and then link to it. The message TEST displays and the program ends.

You've just loaded an assembly language program completely from BASIC.

Using *Mini Memory*'s 4K

When you're working on an assembly language program, you have only about 760 bytes of the module's 3800 bytes to work with. This is because the *Assembler* uses up most of the cartridge's available memory.

Fortunately, an assembly language program doesn't need the *Assembler* once it's ready to run. You can use the 4K RAM of the module for your program if you want. All you have to do is divide your program into segments, each of which fits in the 760 bytes available, and then load these segments, one at a time, from BASIC to any location in the module's RAM.

First, create one segment of the program in the usual memory area starting at >7D00. When it's completed, translate it into its decimal equivalent, just as you saw in the last section. Do this for each segment. Beginning at >7200, for example, load the segments, one after the other. Load each so that it's located immediately after the previous segment. If one ends at >7940, the next should be loaded at the next available address, probably >7942 (it depends on the last instruction used in the first segment).

When all the parts have been loaded, update the LFAM to the address where the REF/DEF Table begins. The entry address and name should point to the first segment. Just run it in the usual manner.

With this technique, you can use the memory occupied by the *Assembler*. However, once you've changed the values of addresses normally used by the *Assembler*, the *Assembler* won't work correctly unless you reload it from tape.

Logical Instructions

Logical instructions come in handy when bits in a byte or word need to be changed. Depending on the circumstances (change some bits but not others, or change all bits), you can select an appropriate instruction. Some of these logical instructions you've already seen. They are:

Instruction		**Function**
CLR	CLeaR	Resets all the bits in a word.
SRL	Shift Right Logical	Moves all the bits in a word a specified number of positions right. Vacant bits are replaced by zeros.
SLA	Shift Left Arithmetic	Moves all the bits in a word a specified number of positions left. Vacant bits are replaced by zeros.

There are other logical instructions, of course. Here they are, along with explanations and short examples of how they can be used.

ORI (OR Immediate). This instruction is used with a register as first operand and an immediate value as second operand. The result of the operation is placed in the register. ORI causes the bits set in *either* of the operands to also be set in the new memory word. For example, if the following words are compared:

```
    0100101110010110
    1001010110101111
```

the resulting value after ORI is:

```
ORI 1101111110111111
```

How about another example? The instruction:

ORI R7,>FF00

lets you know that—whatever the value in R7—the result of the ORI will have the eight left bits set (because >FF is a byte will all its bits set). If R7 is loaded with >1803, for instance, then:

```
 >1803  =  0001100000000011
 >FF00  =  1111111100000000
   ORI  =  1111111100000011
```

The value now in R7 is >FF03 (65283 decimal).

ANDI (AND Immediate). The ANDI instruction is used in the same way as ORI, but only the bits set in *both* words will be set in the new word. All other bits will be reset.

```
         1011011100010011
         0111000110010101
  ANDI   0011000100010001
```

The instruction ANDI R7,>000F keeps the right four bits of the word in R7 unchanged. All other bits are reset. If R7 is loaded with >FF05, then

```
  >FF05  1111111100000101
  >000F  0000000000001111
   ANDI  0000000000000101
```

The new value in R7 is >0005.

XOR (eXclusive OR). This instruction is used with a register or memory address as first operand and a register as second operand. The result of the operation is placed in the second operand (the register). XOR compares the bits of the word in the first operand to the bits of the word in the second operand. The bits which are reset (0) in both words are left reset. Bits set in both words are reset in the new word. Finally, the bits set in one word, but reset in the other, are left set in the resulting word. Sounds complicated, but it's really quite simple. Just look at the example below.

```
        1010001110111011
        1100011101100111
  XOR:  0110010011011100
```

INV (INVert). INV inverts the condition of each bit in a word. If the bit was reset, it's then set. If it was set, then it's reset. With the memory word below:

```
        1011011000010101
```

INV will change it to:

```
        0100100111101010
```

For example, the instructions

INV R7
INV @>7F00

would change the condition of every bit in the memory word at R7 and the word at location >7F00.

SETO (SET to One). This causes *all* the bits of the operand to be set. The instructions

SETO R9
SETO @NM

would leave the words at R9 and at label NM with the value of >FFFF.

SOC (Set Ones Corresponding). SOC sets the bits in the second operand which are set in the first operand. Set bits in the second operand which correspond to reset bits in the first operand remain set. In other words, if a bit is set in either of the two operands, it ends up set in the result. Consider the following two memory words:

```
        0100110100011110
        0011010100101101
```

The SOC instruction would result in:

```
        0111110100111111
```

The instruction uses registers or memory addresses as operands, and the result is left in the second operand.

SOC R7,@>7FE8

The above instruction, for instance, compares the value at R7 with the value at address >7FE8. If the value at R7 is >7A18 (31256 decimal) and the value at address >7FE8 is >00E8 (232 decimal), and the two are compared with SOC, the result placed in location >7FE8 would be calculated by:

```
         0111101000011000 (>7A18)
         0000000011101000 (>00E8)
    SOC  0111101011111000
```

The result is >7AF8 (31480 decimal).

SOCB (Set Ones Corresponding, Byte). It has exactly the same effect as the SOC instruction, except that the right bytes of each word are left unchanged.

SOCB R7,R8

The instruction above sets bits in the left byte of R8 which correspond with bits set in the left byte of R7. Bits set in the left byte of R8 which correspond to reset bits in the left byte of R7 remain set. The right bytes of both words remain unchanged.

If R7 held >7A18 and R8 held >00E8,

```
         0111101000011000 (>7A18)
         0000000011101000 (>00E8)
   SOCB  0111101011101000
```

then the result placed in R8 would be >7AE8.

SZC (Set Zeros Corresponding). This instruction resets bits in the second operand which correspond to set bits in the first operand. Bits set in the second operand which correspond to bits reset in the first operand remain set.

```
         1001001101110111 (>9377)
         0110100110001001 (>6989)
    SZC  0110100010001000
```

places >6888 in R8.

SZCB (Set Zeros Corresponding, Byte). This has the same effect as SZC, but only operates with the *left* bytes of the two words.

SRA (Shift Right Arithmetic). SRA moves every bit of the word in the first operand a specified number of positions to the right. The vacant bits, instead of being set to zero as with SRL, are set equal to the sign bit, which is the leftmost bit of the word to be operated with. If the word value is positive, the vacant positions are reset. If the value is negative, the vacant positions are set. (Remember that any value which has a decimal equivalent greater than 32767 becomes a negative number.) The instruction operates with a register as first operand (where the result of the operation is placed) and an immediate value as second operand.

The second operand can be 0. If that's the case, the word in the first operand is shifted right the number of bits equal to the value of the four least significant bits held in R0. In hexadecimal, that would be the rightmost digit of the value in R0. If the value is 0, the bits are shifted 16 positions.

The instruction

SRA R1,4

would shift the word in R1 four bits to the right. If R1 contained

```
1000100100001111
```

the instruction would result in

```
1111100010010000
```

because the bits are all shifted four to the right. The vacant bits are filled with 1's, since that's the sign bit (note that a 1 is the leftmost bit of the memory word held in R1).

SRC (Shift Right Circular). This instruction moves every bit of the word in the source operand a specified number of positions to the right. The vacant bits are replaced by the bits which have moved out, so to speak, from the right side of the word. It's as if the bits wrap around. This instruction also operates with a register as first operand (where the result of the operation is placed) and an immediate value as second operand.

If, for instance, you used the instruction:

SRC R1,4

and R1 contained this word:

```
0111000101010001
```

the SRC instruction would place this word in R1:

```
0001011100010101
```

Notice that the four rightmost bits (0001) were pushed off the word and wrapped around to the left side.

Displaying the Value in a Register

In many programs you'll want to display the value stored in a register. You'll probably want it shown in decimal.

The following method lets you do so. The hexadecimal value is first divided by ten. If you call Q the answer and R the remainder, then R is the unit digit of the decimal equivalent.

In general, if X is the hexadecimal value (between >0000 and >FFFF), then the decimal equivalent is found by:

```
 X/10 = Q1, with remainder R1
Q1/10 = Q2, with remainder R2
Q2/10 = Q3, with remainder R3
Q3/10 = Q4, with remainder R4
Q4/10 = Q5, with remainder R5
```

The decimal number is then formed by the digits R5-R4-R3-R2-R1.

The following assembly language program assumes that the value you want to display is found in R3. This value is then divided by ten and the remainder placed on the screen (48 is added to the remainder—this gives the ASCII code of the digit which is the remainder). The screen displaying position is then decreased for the next digit, and the procedure is repeated five times (the maximum decimal number that can be represented in hexadecimal by one memory word is only five digits long).

Hex to Decimal

The following example demonstrates this technique. It clears R3 and then displays the value on the screen. The value in R3 is incremented by one and displayed again. This goes on until the value in R3 is >FFFF. The sequence then repeats.

```
7D00        LWPI >70B8    (Memory area for registers)
7D04        CLR R3        (Value to be displayed in R3. Start
                          with zero)
7D06   LP   BL @DP        (Branch to the displaying
                          subroutine)
7D0A        INC R3        (Increase value to be displayed by
                          one)
```

```
7D0C        JMP LP          (Return to the displaying loop)
7D0E   DP   LI R0,305       (Start the subroutine to display
                            the value in the register. The last
                            digit of the number will be dis-
                            played at screen position 305)
7D12        MOV R3,R7       (Move the value to be displayed
                            to R7)
7B14   ST   LI R1,10        (Start the loop to display the dig-
                            its. Load the value to divide with
                            in R1)
7D18        CLR R2          (Clear R2, the left word of the
                            two-word value which will be di-
                            vided by ten)
7D1A        DIV R1,R2       (Divide the value by ten. The
                            remainder of the division, the
                            value to be displayed, will be left
                            in R3)
7D1C        MOV R3,R1       (Move the remainder to R1)
7D1E        AI R1,48        (Add 48 to get to the correspond-
                            ing ASCII code of the remainder)
7D22        SWPB R1         (Place it in the left byte of R1 for
                            the VSBW utility)
7D24        BLWP @>6024     (Display the digit)
7D28        DEC R0          (Decrease screen position for the
                            next digit)
7D2A        CI R0,300       (Has position 300 been reached?
                            That would mean the five digits
                            have been printed on the screen)
7D2E        JNE NF          (If not, jump to NF)
7D30        MOV R7,R3       (Move the value from where it
                            had been stored [R7] back to R3)
7D32        B *R11          (Return from the subroutine to the
                            main control loop)
7D34   NF   MOV R2,R3       (Prepare for the next digit to be
                            displayed. Move the integer result
                            of the division from R2 to R3 for
                            the next division)
7D36        JMP ST          (Jump back to divide the new
                            value)
```

For negative values, you can check the leftmost bit of the word value. If it's set (1), the number is negative and you must print a minus sign before the number. You can calculate the decimal value as you did earlier. Remember that in this case, the values that can be represented by a memory word range from −32768 to 32767.

Using this short routine, you can display scores of games, answers to calculations, and almost any other number stored in a workspace register.

The Screen Buffer

While you're writing your assembly language programs, you can review what you have written by using the up- and down-arrow keys (E and X keys respectively). You can do this because whatever is written to the screen is stored by the computer in a memory area which has space for nine complete screens of text. Once the end of the reserved area is reached, new screens of text are stored at its beginning. What this means is that when you're writing long programs, you'll suddenly find the *Line-by-Line Assembler* title screen reappearing. If you check the memory address you're at, you'll find that you're still in the correct place. Part of what you wrote when you started out your program will appear. Write your new instructions over it, just as if it weren't there. Be careful, since it's easy to get confused when the new instructions become mixed with the old.

Random Numbers

Here are two methods you can use to create random numbers in your program. If your assembly language program will be called from BASIC and you need only a relatively short list of values to be used just once, you can generate the values in BASIC and POKE them into memory. If ten random numbers have to be used in the assembly language program, for example, and these values have to be between 0 and 12, you can, from BASIC, create these and POKE them to a free memory area. The following program does that, and loads the values to CPU RAM starting at address >7F00 (32512 decimal).

```
100 MEM=32512
110 RANDOMIZE
120 FOR A=1 TO 10
13 X=INT(RND*12)+1
140 CALL LOAD(MEM,X)
150 MEM=MEM+1
160 NEXT A
```

This program loads ten random numbers between 0 and 12 into memory addresses >7F00 through >7F09.

This method is useful when an assembly language program uses only a few values, and then uses them only once. When you constantly need to generate random numbers and your program runs entirely in assembly language, you can use the random number in address >83C0. This location contains a different word value each time the computer is reset (with FCTN = (QUIT), or by turning the computer off).

Try the following: Insert the *Mini Memory* cartridge and select EASY BUG. Skip the title screen and type M83C0 to see the contents of address >83C0. Look at the contents of the address's least significant byte(>83C1), too, by pressing ENTER. Make a note of the word value. Then press FCTN = (QUIT) and repeat the procedure. Each time you do this, the word value at >83C0 is different. Now let's see how this can help, since you can't be expected to constantly reset the computer each time you need a random number.

Imagine the value placed at >83C0 as being a memory address. The value stored at that memory address will be your first random number (between >0000 and >FFFF). For the next random number, add any number (>1254, for instance) to the memory address. This gives you a new address. The value there will be your new random number, and so on. As you have no way of knowing your initial memory address, the numbers generated will be random.

This procedure creates numbers between >0000 and >FFFF. For smaller values, shift the value in the register with the SRL instruction, which fills vacant positions with zeros. With a comparison instruction, you can ignore values you don't want to accept, sending control back to look for a new random number. See the following table for a list of the random values you can expect when you use various SLA instructions.

Instruction	**Random Value Range**
No shifting	0–65535
SRL RX,1	0–32767
SRL RX,2	0–16385
SRL RX,3	0–8191
SRL RX,4	0–4095
SRL RX,5	0–2047
SRL RX,6	0–1023

SRL RX,7	0–511
SRL RX,8	0–255
SRL RX,9	0–127
SRL RX,10	0–63
SRL RX,11	0–31
SRL RX,12	0–15
SRL RX,13	0–7
SRL RX,14	0–3
SRL RX,15	0–1

The following program calculates a random number between 0 and 767 (one number for each possible screen position) and prints an asterisk at that address. This will continue until 300 asterisks have been printed (the entire screen won't be filled). The program then stops.

```
7D00        LWPI >70B8        (Load memory area for the
                               registers)
7D04        MOV @>83C0,R3     (Move random initial value to R3)
7D08        CLR R15           (Clear R15, which keeps track of
                               the asterisks printed)
7D0A  RN    MOV *R3,R5        (Move the value stored at the ad-
                               dress in R3 to R5)
7D0C        SRL R5,6          (Make it a random value between
                               0 and 1023. Screen goes up to 767
                               and starts at 0)
7D0E        CI R5,768         (Check whether the random num-
                               ber is between 0 and 767)
7D12        JGT CT            (If it's greater, jump to CT, where
                               the program gets ready to choose
                               a new random number)
7D14        JMP PR            (Jump to print the asterisk)
7D16  GT    AI R3,>1218       (Change the address to select a
                               new number)
7D1A        JMP RN            (Jump back to select a new ran-
                               dom number)
7D1C  PR    MOV R5,R0         (Move the asterisk's screen po-
                               sition to R0 for the VSBW utility)
7D1E        LI R1,>2A00       (Load the code for the asterisk in
                               R1)
7D22        BLWP @>6024       (Print the asterisk on the screen)
7D26        INC R15           (Increase the number of asterisks
                               printed)
7D28        CI R15,300        (Have 300 asterisks been printed?)
```

7D2C	JNE CT	(If not, return for a new random screen position)
7D2E	JMP $	(Stop the program with an endless loop)

Now run the program and see how quickly the asterisks are printed.

Your New Language

Assembly language programming on the TI isn't as mysterious as you might have thought, is it? With patience, some time, and the right guide, you can explore this language's potential. That's what you've been doing all through this book.

You've seen just some of the possibilities of assembly language programming, though. More complex programs can be created. Experiment, test, and toy with short routines, gradually developing them until you're ready to splice them together. Before you realize it, you'll be designing and writing longer programs which take full advantage of your computer's capabilities.

Like any new language, assembly language takes time to learn. You get better at it with practice. Fortunately, with this guide in hand, you've got a strong foundation.

Decimal and Hexadecimal ASCII Codes

Character	Decimal ASCII Code	Hexadecimal ASCII Code
NUL	0	00
SOH	1	01
STX	2	02
ETX	3	03
EOT	4	04
ENQ	5	05
ACK	6	06
BEL	7	07
BS	8	08
HT	9	09
LF	10	0A
VT	11	0B
FF	12	0C
CR	13	0D
SO	14	0E
SI	15	0F
DLE	16	10
DC1	17	11
DC2	18	12
DC3	19	13
DC4	20	14
NAK	21	15
SYN	22	16
ETB	23	17
CAN	24	18
EM	25	19
SUB	26	1A
ESC	27	1B
FS	28	1C
GS	29	1D
RS (cursor)	30	1E
US (edge)	31	1F
Space	32	20

Character	Decimal ASCII Code	Hexadecimal ASCII Code
!	33	21
"	34	22
#	35	23
$	36	24
%	37	25
&	38	26
'	39	27
(	40	28
)	41	29
*	42	2A
+	43	2B
,	44	2C
−	45	2D
.	46	2E
/	47	2F
0	48	30
1	49	31
2	50	32
3	51	33
4	52	34
5	53	35
6	54	36
7	55	37
8	56	38
9	57	39
:	58	3A
;	59	3B
<	60	3C
=	61	3D
>	62	3E
?	63	3F
@	64	40
A	65	41
B	66	42
C	67	43
D	68	44

Character	Decimal ASCII Code	Hexadecimal ASCII Code
E	69	45
F	70	46
G	71	47
H	72	48
I	73	49
J	74	4A
K	75	4B
L	76	4C
M	77	4D
N	78	4E
O	79	4F
P	80	50
Q	81	51
R	82	52
S	83	53
T	84	54
U	85	55
V	86	56
W	87	57
X	88	58
Y	89	59
Z	90	5A
[	91	5B
\	92	5C
]	93	5D
^	94	5E
_	95	5F
`	96	60
a	97	61
b	98	62
c	99	63
d	100	64
e	101	65
f	102	66

Character	Decimal ASCII Code	Hexadecimal ASCII Code
g	103	67
h	104	68
i	105	69
j	106	6A
k	107	6B
l	108	6C
m	109	6D
n	110	6E
o	111	6F
p	112	70
q	113	71
r	114	72
s	115	73
t	116	74
u	117	75
v	118	76
w	119	77
x	120	78
y	121	79
z	122	7A
{	123	7B
¦	124	7C
}	125	7D
~	126	7E
DEL	127	7F

Assembly Language Instructions

This list includes all the instructions which you can use with the *Line-by-Line Assembler.* The first column indicates the instruction's *mnemonic,* the four-character (or less) word which represents the instruction. The second column specifies the instruction's name, and the third column shows the format to which it belongs.

Mnemonic	Instruction	Format
A	Add words	I
AB	Add Bytes	I
ABS	ABSolute value	VI
AI	Add Immediate	VIII
ANDI	AND Immediate	VIII
B	Branch	VI
B *R11	Same as ReTurn (RT)	VI
BL	Branch and Link	VI
BLWP	Branch and Load Workspace Pointer	VI
C	Compare words	I
CB	Compare Bytes	I
CI	Compare Immediate	VIII
CKOF	ClocK OFf	VII
CKON	ClocK ON	VII
CLR	CLeaR	VI
COC	Compare Ones Corresponding	III
CZC	Compare Zeros Corresponding	III
DEC	DECrement	VI
DECT	DECrement by Two	VI
DIV	DIVide	IX
IDLE	IDLE	VII
INC	INCrement	VI
INCT	INCrement by Two	VI
INV	INVert	VI
JEQ	Jump if EQual	II
JGT	Jump if Greater Than	II
JH	Jump if logical High	II
JHE	Jump if High or Equal	II
JL	Jump if logical Low	II
JLE	Jump if Low or Equal	II

Mnemonic	Instruction	Format
JLT	Jump if Less Than	II
JMP	Unconditional JuMP	II
JNC	Jump if No Carry	II
JNE	Jump if Not Equal	II
JNO	Jump if No Overflow	II
JOC	Jump On Carry	II
JOP	Jump if Odd Parity	II
LDCR	LoaD CRU	IV
LI	Load Immediate	VIII
LIMI	Load Interrupt Mask Immediate	VIII
LREX	Load or Restart EXecution	VII
LWPI	Load Workspace Pointer Immediate	VIII
MOV	MOVe word	I
MOVB	MOVe Byte	I
MPY	MultiPlY	IX
NEG	NEGate	VI
NOP	No OPeration	II
ORI	OR Immediate	VIII
RSET	ReSET	VII
RTWP	ReTurn with Workspace Pointer	VII
S	Subtract words	I
SB	Subtract Bytes	I
SBO	Set CRU Bit to One	II
SBZ	Set CRU Bit to Zero	II
SETO	SET to One	VI
SLA	Shift Left Arithmetic	V
SOC	Set Ones Corresponding	I
SOCB	Set Ones Corresponding, Byte	I
SRA	Shift Right Arithmetic	V
SRC	Shift Right Circular	V
SRL	Shift Right Logical	V
STCR	STore CRU	IV
STST	STore STatus	VIII
STWP	STore Workspace Pointer	VIII
SWPB	SWaP Bytes	VI
SZC	Set Zeros Corresponding	I

Mnemonic	Instruction	Format
SZCB	Set Zeros Corresponding, Byte	I
TB	Test Bit	II
X	EXecute	VI
XOP	EXtended OPeration	IX
XOR	EXclusive OR	III

Courtesy of Texas Instruments, Incorporated

Distinguishing Operands

Here's a list of all the instructions you can use with the *Line-by-Line Assembler,* together with the operands each instruction requires.

When only one operand is used, a dash (—) is placed in the third column. A dash can also indicate that the instruction does not place a result in an operand, such as in the comparison instructions. If no operands are required, a dash is also placed in the second column. Determine the operands to use according to the following table:

R =	Register
N =	Decimal or hexadecimal number
GA =	A general address (register, memory address, etc.)
MA =	A memory address (a decimal or hexadecimal number, a label, etc.)
CA =	A CRU (Control Register Unit) bit address
(*) =	The decimal number is between 0 and 15

Mnemonic	Operands	Operand Where the Result Is Placed
A	GA,GA	Second
AB	GA,GA	Second
ABS	GA	—
AI	R,N	First
ANDI	R,N	First
B	GA	—
BL	GA	—
BLWP	GA	—
C	GA,GA	—
CB	GA,GA	—
CI	R,N	—
CKOF	—	—
CKON	—	—
CLR	GA	—
COC	GA,R	—
CZC	GA,R	—
DEC	GA	—
DECT	GA	—
DIV	GA,R	Second
IDLE	—	—
INC	GA	—
INCT	GA	—
INV	GA	—

Mnemonic	Operands	Operand Where the Result Is Placed
JEQ	MA	—
JGT	MA	—
JH	MA	—
JHE	MA	—
JL	MA	—
JLE	MA	—
JLT	MA	—
JMP	MA	—
JNC	MA	—
JNE	MA	—
JNO	MA	—
JOC	MA	—
JOP	MA	—
LDCR	GA,N (*)	—
LI	R,N	First
LIMI	N	—
LREX	—	—
LWPI	N	—
MOV	GA,GA	Second
MOVB	GA,GA	Second
MPY	GA,R	Second
NEG	GA	—
NOP	—	—
ORI	R,N	First
RSET	—	—
RTWP	—	—
S	GA,GA	Second
SB	GA,GA	Second
SBO	CA	—
SBZ	CA	—
SETO	GA	—
SLA	R,N (*)	First
SOC	GA,GA	Second
SOCB	GA,GA	Second
SRA	R,N (*)	First
SRC	R,N (*)	First
SRL	R,N (*)	First
STCR	GA,N (*)	First
STST	R	—
STWP	R	—
SWPB	GA	—
SZC	GA,GA	Second
SZCB	GA,GA	Second

Mnemonic	Operands	Operand Where the Result Is Placed
TB	CA	—
X	GA	—
XOP	GA,N (*)	—
XOR	GA,R	Second

Courtesy of Texas Instruments, Incorporated

Appendix D

Color Codes

Hexadecimal Color Codes

Code	Color
0	Transparent
1	Black
2	Medium Green
3	Light Green
4	Dark Blue
5	Light Blue
6	Dark Red
7	Cyan
8	Medium Red
9	Light Red
A	Dark Yellow
B	Light Yellow
C	Dark Green
D	Magenta
E	Gray
F	White

Index